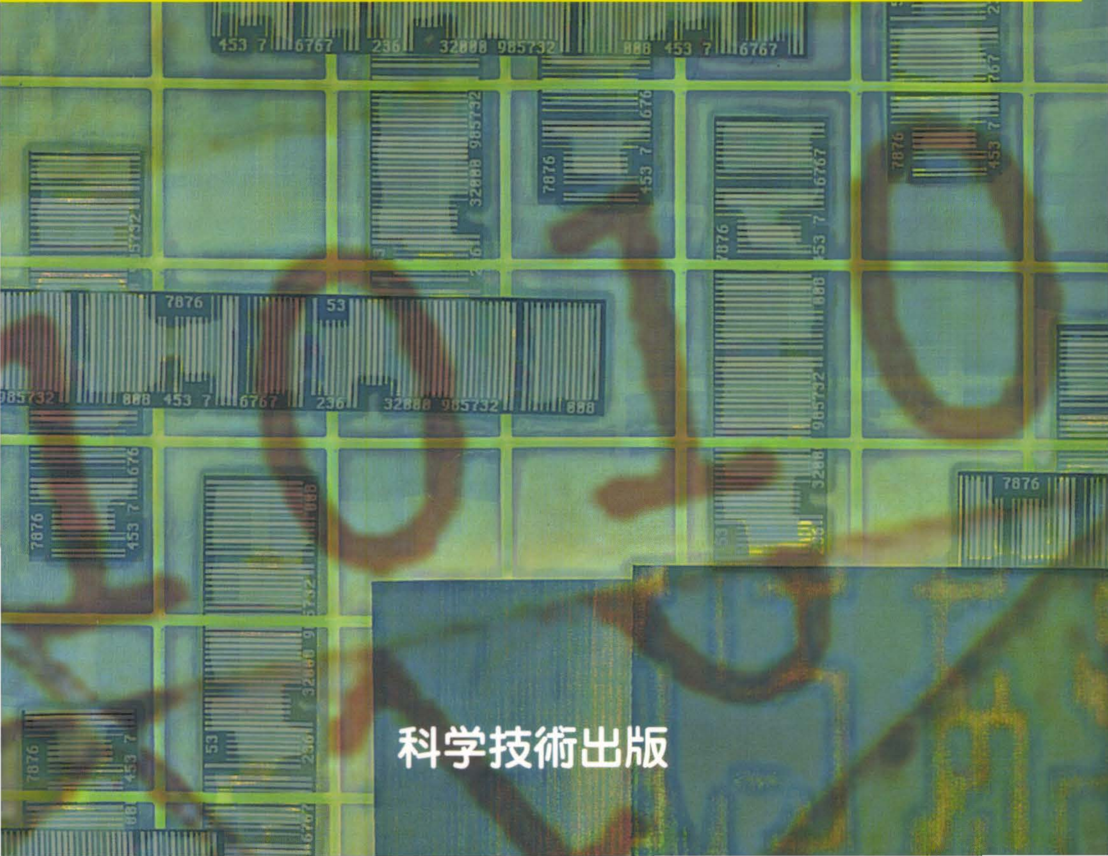
コンピュータサイエンス・シリーズ

システム・プログラム

ジョン・J・ドノバン…………［著］／池田克夫…………［訳］



科学技術出版

コンピュータサイエンス・シリーズ

システム・プログラム

ジョン・J・ドノバン 著
池田克夫 訳

科学技術出版

Systems Programming

by John J. Donovan

訳　者　序　文

コンピュータで仕事を処理するためには，プログラミング言語を設計して，その言語処理プログラムを実現することに始まり，ソースプログラムのコーディングと実行プログラムへの変換，最後に実行プログラムをコンピュータの中へ読み込んで実行するという過程を経る必要がある．

一般のプログラマにとって，これらの過程を全て知る必要はない．しかし，情報科学・工学を学ぶ人たちにとって，このプログラミング環境の原理と構成を理解することは必須である．

ところで，コンパイラの理論やコンパイラコンパイラのような高水準言語の処理に関する教科書は，これまでに多数刊行されている．しかし，目的プログラムの生成とその実行に至るまでのライブラリの結合からプログラムのロードまでの過程を，専門課程学生の学習に適するように，丁寧にアルゴリズムを示して説明している教科書はほとんどなかったのである．

訳者は，MIT に客員研究員として滞在していたときに J. J. Donovan の教科書「システムプログラミング」に出会い，教科書としての内容の適切さを知り，翻訳出版して，京都大学の情報工学科において講義に使用していた．

その後，しばらくシステムプログラムの講義を担当することがなかったが，この度再び担当することになったので，手に入る教科書を全てチェックしてみた．ところが，Donovan の教科書の右に出る教科書が見つからなかったのである．

本書は IBM 370 のアーキテクチャに従って書かれている．この四半世紀の間にコンピュータは驚くほど進歩した．それは LSI の集積密度と動作速度の飛躍的な向上，およびネットワーク機能であった．コンピュータアーキテクチャの観点からは，並列処理機能と高速アクセスを可能にするメモリアーキテクチャの工夫，ネットワークによる分散処理機能に留意しなければならない．その結果，オペレーティングシステムの構成には大きな影響があった．しかし，プログラミング環境に関して，目的プログラムの生成と実行という面からは特段に注意を払うべきものはなかったのである．

この点からは，IBM 370 は完成したハードウエアであったと評価できる．その後 CISC，RISC のアーキテクチャが多数採用されているが，370 のアーキテ

クチャを理解しておけば，これらの理解に苦労することはない．プログラムの目的コードの生成についても同様である．むしろ教科書に取り上げる場合には，本質的な事項の理解を重視するために，CPU の基本機能に主眼を置いて，特殊な機能は極力省くことが必要である．このような訳で，CISC や RISC になっても，プログラミング環境の学習のためには，付け加えることはなかった．

一方，最近多用されている Java について考えてみると，Java プログラムを実行するバーチャルマシンはスタックマシンであり，簡単すぎて，これで全てのことを代表することはできない．全ての CPU がスタックマシンに変わってしまうということでもなければ，オペレーティングシステムの実行プログラムを構成するための勉強にはならないと考えられる．

本書の主旨が理解されて情報科学・工学の基礎の学習に寄与できれば幸甚である．

2002 年 2 月　京都衣笠にて

池田　克夫

序　文

目　的

本書では，アセンブラ，マクロ，ローダ，コンパイラおよびオペレーティング・システムの使用とインプリメンテーションを含め，システム・プログラム全般にわたって述べようとしている．すなわち，新しい計算機用言語と新しいオペレーティング・システムに関連して，その構成要素を詳細に示し，その設計の問題点を明らかにする．これらの問題点としては構文や意味論といった従来のコンパイラの問題に加えて，記憶装置割当てや競合の生じる参照の問題がある．また今日では，ファイル・システム，多重処理，多重プログラミングなどがオペレーティング・システムにおいて普通のこととなっている．計算機科学のより公式的な面を紹介するために，形式的体系による表現とプログラム言語への応用をも含めた．

本書は教科書として，演習問題に特に重点を置いて書いてある．読者は高水準言語での経験を持っているものと仮定している．

本書の内容は可能な限り機械と独立にするよう努力しており，実際に，幾種かの違った型の計算機を用いた教育に使用されている．しかしながら，真実味を増すためにIBM 360/370 型の計算機を具体例とし，またコンパイラの議論にはPL/I に示される特徴を持った言語を例として用いている．

本書はACM（Association for Computing Machinery）の計算機科学におけるカリキュラム委員会[1] が提示したカリキュラム68の6つの科目の内容を網羅している．基礎科目である計算機の構成とプログラミング（B 2）の内容は第1章から第5章に，プログラム言語（12）とコンパイラの構成（15）の内容は第6章，第7章および第8章に，システム・プログラム(14)，進んだ計算機構成(A 2)，および，大規模情報システム（A 8）の内容は第9章において扱っている．

---

1 CACM vol. 11, no. 3, p. 151 (March 1968) 参照．

本書の使用法

本書は主として次の3つの使用法がある．(1)1～2学期分のシステム・プログラムの学部向き科目の教科書として，(2)専門家用として，(3)大学院学生の参考書として．

具体的には本書は次の科目の要求にかなうものとして使用されている：

1. （例えば FORTRAN や PL/I のプログラム講習的な科目に引続く）計算機科学の学部教育の最初の科目
2. 計算機関係以外の一般教育科目
3. ソフトウェアの上級科目
4. 実際的な問題に重点を置くソフトウェア工学の科目
5. 計算機科学の大学院生に対する概論または導入的科目

MIT (Massachusetts Institute of Technology：マサチューセッツ工科大学）において，本書はディジタル計算機プログラミング・システム 6.251 という学部向き科目で使用されている．この科目には伝統があり，学期毎に 350 人が選択し，最も学生数の多い選択科目の1つである．また，この科目は最もやりがいのある科目の1つでもある．MIT でこの科目は上述の目的により開設されている．

また，同じ教材は2学期間の大学院科目にも使用されている．最初の学期では計算機の構成，アセンブラ，マクロ，ローダ，I/O プログラミングおよびオペレーティング・システムの話題が取り扱われる（第1，第2，第3，第4，第5章，および第9章の一部分)．次の学期ではプログラム言語，コンパイラの設計，形式的体系，および最初の学期で扱われなかったオペレーティング・システムの他の機能が論じられる（第6，第7，第8，および第9章)．

同じ科目は Honeywell，米国水底システム・センター (U. S. Underwater System Center)，Martin Marietta，その他の幾つかの企業で，主としてシステムの構成要素の設計問題に焦点を合わせ，形式的体系（第7章）を省いて教

えられている.

この教科書はフランスの SESA の上級プログラムの科目や，Boston の Lowell School の2年間の技術者養成課程の科目でも使用されていた.

Lubbock, Texas の Texas 工科大学でのプログラムの科目は，最初 MIT の講義のビデオ・テープを用いて行なわれた. MIT のビデオ・テープは次の週に Texas に送られて再生された. このようにして授業を他で行なうことは，Texas の学生と MIT の学生に試験で同じ問題を出題し，成績に目立った差がなかったので，有効であることが実証された.

MIT でこの科目に関連して使用された機械は IBM 360/370 型計算機である. 米国海底音響研究所 (U. S. Underwater Sound Laboratories) では UNIVAC 1108 が使用されていた. Honeywell では各種の Honeywell の機械が使用されていた.

学生がアセンブラ言語や PL/I の経験を持っていることは内容を理解する上で役に立つ. しかしこれらの経験を持っていない場合には，計算機のマニュアル，講義，および本書の第2章と第6章をアセンブラ言語や PL/I 入門の手引きとして使用することができる.

専門家または大学院学生が使用するとき，本書には IBM 370 と PL/I に関して十分な内容が含まれており，残りの部分を読むために他の本を参照しなくてもよい.

アセンブラやマクロなど従来のシステム要素の議論に加えて，本書は現在発行されている教科書にはまだあまり述べられていないシステム・プログラムの重要な特徴，コンパイラや記憶装置割付け，回帰呼出し，オペレーティング・システムおよび I/O プログラミングなどの高度の問題に特に重点を置いている.

謝　　辞

本書自身がこれに寄与した多くの人達の尽力と技術的能力に対する謝辞である．ここでは極く一部分の人々の名前だけしかあげることができない．

Stuart Madnick 氏の貢献は特筆すべきものである．Madnick 氏は私の知っている限り最も優れたシステム・プログラマである．氏は私と一緒に5年間，助手および講師として，本書を使用する MIT の科目（ディジタル計算機プログラミング・システム）の講義を行なった．氏はファイル・システムとオペレーティング・システム概論の部分を受持った．また氏はすべての章を読みその内容に寄与した．私は氏の技術的貢献と忠実さに感謝したい．

MIT のディジタル計算機プログラミング・システムはこれを担当した教授達によって発展してきた．これらの教授の名をここにあげておく

1959年	F. Verzuh 教授
1960年	F. J. Corbató 教授
1961年	W. Poduska および N. Haller 教授
1962年	J. Saltzer 教授
1963年	C. L. Liu, D. Kuck, F. J. Corbató および D. Thornhill 教授
1964年	J. Saltzer, D. Kuck, R. Fabry および T. Stokham 教授
1965年	W. Poduska, L. Hatfield, R. Baecker および R. M. Graham 教授
1966年	R. M. Graham 教授
1967年—現在	J. J. Donovan 教授

科目の内容はアセンブラから始まってローダ，探索法，デバッグ技法，最近になってオペレーティング・システムを含むに至ったので，個々の講師の貢献を厳密に述べるのは困難である．Corbató 教授は統一的な概念を設定し，事例

研究法を導入した．私の前任者 Graham 教授は私に多くの基本事項を教授されたとき，私の考え方に大きな影響を与えた．

過去5年の間，多くの助手の援助によりこの科目を IBM 360 に合せて再構成し，直接結合型のローディング方式，PL/I，高水準コンパイル技法，記憶装置とプロセッサの管理，および，最近ではI/Oプログラミングを組み入れた．

毎年優れた助手のグループがこの科目のために働いてくれたので，350 人もの学生を1学期に教えることができた．学生の広範な要求をさばいてくれたこれら助手各位の尽力と能力に負うところが大きい．過去数年間のこれら助手の長は Stuart Madnick, Steven Zilles, Richard Mulhern, William Michels, および Glen Brunk である．しかし他の多くの人々も様々な方法で寄与し，特に問題の設定に対して尽力してくれた．ここに幾人かの名をあげておく．David Quimby, Jerry Johnson, Michael Hammer, Benjamin Ashton, Orville Dodson, Steven Halfich, Norman Kohn, Martin Jack, Al Moulton, Chander Ramchandani, および Harris Berman. Muriel Webber 夫人は図版の作成をしてくれた．Max Byer は MIT の科目の管理に貢献した．

次の人々の多大な努力が本書の内容に貢献した．William Michels はコンパイラの章に大きく寄与した．Norman Kohn はマクロと形式的体系の章および参考文献の整理に寄与した．Sterling Eanes は形式的体系およびコンパイラの章を校正した．David Quimby は演習問題の編集を行ない，本書の草稿に意見を述べてくれた．Leonard Goodman は本書全部の校正を受持ってくれた．訂正された誤りに対しては氏に礼を述べたい．David M. Jones 氏は原稿を編集してくれた．Ellen Nangle はタイプを指揮し，原稿の準備をしてくれた．彼女はさらに，原稿，演習問題，解答および教師用マニュアルを編集した．

本書の中で報告されている研究の一部は国防省の上級研究計画局（Advanced Research Projects Agency）の後援で，海軍研究契約（Office of Naval Research Contract Nonr-4102(01))によって設立された研究プロジェクトである MIT のプロジェクト MAC において行なわれ，一部は MIT の電気工学科で行なわれた．

最後に，形式的にであると思われそうなのでためらっているのであるが，妻

謝　　辞

と５人の子供達に心から一言ありがとうをいいたい．私の仕事は，過去２年間，１日24時間，１週７日，そして，それ以前も長期間この原稿を用意するのにかかっていたから，家族の忍耐と援助に感謝している．

John J. Donovan

学 生 諸 君 に

本書はアセンブラ言語の深い知識を仮定してはいないが，この言語に経験を有する学生には理解しやすいであろう．第2章はアセンブラ語のプログラミングに関する基本的な事項を，第6章は高水準言語，特に，PL/I について述べている．アセンブラ言語および PL/I のプログラミングに関しては，他の参考書をも参照するのがよい．

第8章および第9章（コンパイラとオペレーティング・システム）は長く内容が盛りだくさんである．（これらの章だけで1つの科目ができるほどである.)これらの章は読者の都合の良いように幾つかの部分に分割してある．

工学の分野において，特に，計算機科学のように急速に発展している分野においては，関係の文献に目を通しておくことが必要である．第10章は参考文献を示している．活用してほしい．

本書はプログラミングを知っていることを前提としている．したがって，未経験の読者にとっては内容が高度で密度が大きく難しいだろうが失望しなくてもよい．

本書の内容は計算機科学の基本である．そしてよく勉強した学生は内容が適切で興味深いと思うだろう．この内容を勉強するためのより公式的な方法を私は知らない．

目　次

訳者序文 ……iii

序　文 ……v

謝　辞 ……viii

学生諸君に ……xi

第1章　背　景 ……1

1.1　計算機の構造 ……2

1.2　プログラミング・システムの構成要素の発展 ……4

1.2.1　アセンブラ ……4

1.2.2　ローダ ……5

1.2.3　マクロ ……7

1.2.4　コンパイラ ……8

1.2.5　形式的体系 ……8

1.3　オペレーティング・システムの発展 ……9

1.4　利用者から見たオペレーティング・システムの機能 ……11

1.5　利用者から見たオペレーティング・システムのバッチ制御言語 ……13

1.6　利用者から見たオペレーティング・システムの設備 ……16

1.7　まとめ ……17

演習問題 ……18

第2章　計算機の構造，機械語およびアセンブラ語 ……24

2.1　一般的な計算機の構造 ……24

2.1.1　新しい計算機を知るための一般的な方法 ……26

2.1.2　計算機の構造——IBM 360と370 ……28

1.　記憶装置 ……28

2.　レジスタ ……29

3.　データ ……30

4.　命　令 ……34

5. 特殊機能 ……36
2.2 機械語 ……39
2.2.1 長い方法――ループなし ……41
2.2.2 命令をデータとして扱う番地修飾法 ……42
2.2.3 インデックス・レジスタを用いた番地修飾法 ……44
2.2.4 ループの構成 ……46
2.3 アセンブラ語 ……47
2.3.1 アセンブラ語のプログラム ……48
2.3.2 リテラルの使用例 ……50
2.4 まとめ ……52
演習問題 ……52

第3章 アセンブラ ……64

3.1 一般的な設計手順 ……65
3.2 アセンブラの設計 ……65
3.2.1 問題の設定 ……65
3.2.2 データ構造 ……69
3.2.3 データ・ベースの書式 ……72
3.2.4 アルゴリズム ……79
3.2.5 機能単位を探すこと ……84
3.3 表の処理：探索と分類 ……87
3.3.1 逐次探索法 ……88
3.3.2 2分探索法 ……89
3.3.3 分類法 ……91
3.3.3.1 互換分類法 ……92
3.3.3.2 Shell分類法 ……93
3.3.3.3 基数分類法 ……95
3.3.3.4 基数交換分類法 ……95
3.3.3.5 番地計算分類法 ……97
3.3.3.6 分類法の比較 ……98
3.3.4 ハッシュ法による登録と探索 ……99
3.4 まとめ ……103
3.5 あとがき ……104

演習問題 …… 105

第4章 マクロ言語とマクロ・プロセッサ …… 121

4.1 マクロ命令 …… 122
4.2 マクロの機能 …… 124
4.2.1 マクロ命令の引数 …… 124
4.2.2 条件付きマクロ展開 …… 128
4.2.3 マクロの中でのマクロ呼出し …… 129
4.2.4 マクロを定義するマクロ命令 …… 131
4.3 組込み …… 132
4.3.1 限定された機能の組込み：2パス・アルゴリズム …… 134
4.3.2 1パスのアルゴリズム …… 141
4.3.3 マクロの中でのマクロ呼出しの組込み …… 145
4.3.4 アセンブラの中への組込み …… 154
4.4 まとめ …… 155
演習問題 …… 155

第5章 ローダ …… 162

5.1 ローダの方式 …… 163
5.1.1 “コンパイル・実行”型ローダ …… 163
5.1.2 一般的なローダの方式 …… 164
5.1.3 絶対型ローダ …… 166
5.1.4 サブルーチンの結合 …… 168
5.1.5 再配置型ローダ …… 170
5.1.6 直接結合型ローダ …… 174
5.1.7 他のローダ方式――バインダ，リンク・ローダ，オーバレイ，ダイナミック・バインダ …… 179
5.2 絶対型ローダの設計 …… 183
5.3 直接結合型ローダの設計 …… 185
5.3.1 問題の設定 …… 186
5.3.2 データ構造の規定 …… 192
5.3.3 データ・ベースの書式 …… 196
5.3.4 アルゴリズム …… 199

5.4 まとめ……204
演習問題 ……205

第6章 プログラム言語 ……216

6.1 高水準言語の重要性……217
6.2 高水準言語の特徴……219
6.3 データの型とデータ構造……220
6.3.1 文字列……221
6.3.2 ビット列……222
6.3.3 データ構造……223
6.4 記憶域の割付けと名前の有効範囲……224
6.4.1 記憶域のクラス……224
6.4.2 ブロック構造……225
6.5 アクセスの柔軟性……226
6.5.1 ポインタ……226
6.5.2 ラベル変数とラベル配列……229
6.6 機能のモジュール化……230
6.6.1 手続き……230
6.6.2 回帰呼出し……231
6.7 非同期処理……231
6.7.1 条件処理……232
6.7.2 シグナル……233
6.7.3 多重タスク……234
6.8 拡張性とコンパイル時のマクロ……234
6.9 その他……235
6.10 まとめ……235
演習問題……235

第7章 形式的体系とプログラム言語：序 説……247

7.1 プログラム言語における形式的体系の使用……247
7.1.1 言語の記述……248
7.1.2 直構文コンパイラ……248
7.1.3 構造の複雑さの研究……249

7.1.4 構造解析 …… 249
7.2 形式的定義 …… 250
7.2.1 形式化の方法 …… 250
7.2.2 形式的定義の発展 …… 252
7.3 形式的文法 …… 255
7.3.1 形式的文法の例 …… 256
7.3.2 文の導出 …… 257
7.3.3 文形式と文 …… 258
7.4 言語の階層構造 …… 258
7.5 BACKUS NAUR FORM—BACKUS 標準形—BNF …… 261
7.6 正準体系 …… 262
7.6.1 例：構文の定義 …… 267
7.6.2 翻訳の定義 …… 269
7.6.3 認識と翻訳のアルゴリズム …… 270
7.7 正準体系と形式的体系 …… 275
7.8 まとめ …… 281
演習問題 …… 282

第 1 章

背　　　　景

本書は主として２つの目的を持っている．ソフトウェア・システム設計の手順を教えること，および，ソフトウェア設計における判断の基準を用意することである．この目的を達成するために，システム・プログラムから特定の例を取り上げ，代表的なシステムの構成要素の設計と組込みについて論じている．

システム・プログラムとは何か．読者は計算機をすべての命令に従う動物のように考えるかもしれない．計算機は基本的に金属でできた人間であるともいわれている．逆にいうと人間は血と肉からできた計算機であるというわけである．しかし計算機のことを知るにつれて，計算機は基本的には一定のしかも原始的な命令に従う機械であるということがわかるであろう．

計算機が生れた当初，人々はスイッチが *on* であるとか *off* であるとかいって基本的な命令を計算機に与えていた．しかしすぐにもっと複雑な命令を出したいと望んだ．例えば $x=30\times y$ のとき $y=10$ ならば x は幾らかというようないい方である．今日の計算機はシステム・プログラムの助けなしにこのような言語を理解することはできない．コンパイラ，ローダ，マクロ・プロセッサ，オペレーティング・システムなどといったシステム・プログラムは利用者の要求により良く適合するようにと開発されたものである．さらに，人々はプログラムを書くため技術者の援助を必要とするようになった．

コンパイラは人が使うような言葉を受理して機械語に翻訳するプログラムである．ローダは実行のために機械語のプログラムを用意するシステム・プログラムである．マクロ・プロセッサはプログラマが省略形でプログラムが書けるようにする．オペレーティング・システムとファイル・システムは情報の柔軟な格納や参照を許すものである（図 1.1）．

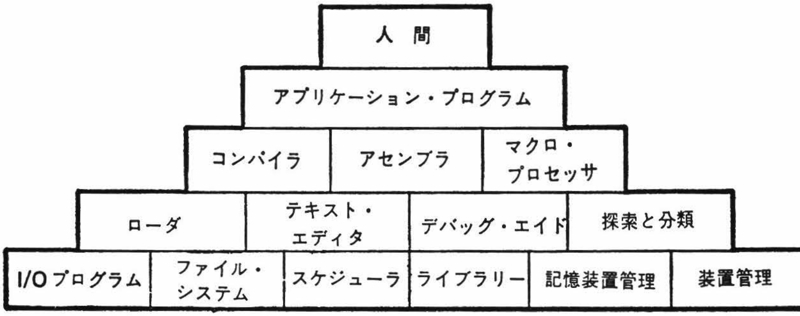


図 1.1　システム・プログラムの構成

今日10万台以上の計算機がありとあらゆる方面に応用されている．各計算機の生産性はシステム・プログラミングの有効性，効率および機能に大きく依存している．

本章では用語と計算機の構造の概要およびオペレーティング・システムの基本的な役割りを紹介する．

1.1　計算機の構造

まず最初に一般的な計算機システムのハードウェア構成を示す（図 1.2）．

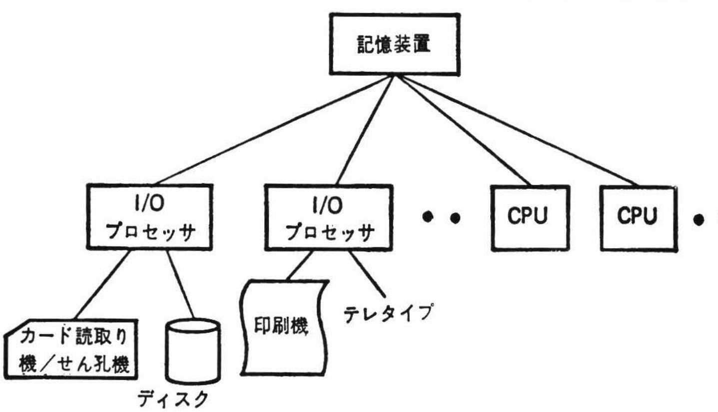


図 1.2　計算機システムの一般的なハードウェア構成

記憶装置は情報が格納される装置である．**プロセッサ**はこの情報に関する処理が行なわれる装置である．情報は1と0の形で格納されている．それぞれの1と0が2進数の1桁に対応し**ビット**と呼ばれる．ビットは語，文字または**バイト**と呼ばれる単位にまとめられる．

記憶されている場所は**番地**によって指定され，各番地は特定のバイト，語または文字を識別する．

語の内容は**データ**（処理される値）または**命令**（実行される処理）として解釈される．プロセッサは記憶装置の中の**命令**によって規定された一連の処理を実行する装置である．**プログラム**（または**手続き**）は一連の命令である．

記憶装置は1と0の集合を納めている郵便箱のように考えることができる．

下に番地が10,000から10,002の記憶装置の内容を示す．

IBM 1130 プロセッサは10,001番地を命令として処理すると，その内容を“停止”命令として解釈する．同じ内容を数値データとして処理すると，プロセッサはその内容を2進数 0011 0000 0000 0000（10進数12,288）と解釈する．このように命令とデータは同じ記憶媒体を共有する．

番地	内容
10,000	0000 0000 0000 0001
10,001	0011 0000 0000 0000
10,002	0000 0000 0000 0100

記憶装置の中の情報は文字，命令または数として解釈されるようなビットの集まりとして符号化される．**符号**（コード code）はビットの集まりの解釈の規則を示すものであり，例えば10進数表現(BCD)のため，文字のため(EBCDIC または ASCII)，または命令（特定のプロセッサの命令）のためのコードがある．前に2種のプロセッサを図で示した．すなわち，**入力／出力**（I/O）プロセッサと**中央演算処理装置**（Central Processing Unit : CPU）である．I/O プロセッサは記憶装置とディスク，ドラム，印刷機およびタイプライタといった周辺装置との間の情報の転送を行なう．CPU は記憶装置に格納されたデータを処理する装置である．I/O プロセッサは記憶装置に格納された I/O 命令を実行する．これらは一般に CPU からコマンドによって起動される．通常コマンドは“I/O 実行”命令から成り，引数は I/O プログラムの開始番地である．CPU はこの命令を解釈して I/O プロセッサに引数を渡す．

I/O 命令は CPU の命令と全く異なったものであり，CPU の動作と**非同期**に（同時に）実行される．I/O チャネルと CPU の非同期動作は最も初期の**多重処理**（multiprocessing）の形態の1つであった．多重処理は2台以上のプロセッサが同じ記憶装置を同時に使用して処理を行なうことをいう．

命令語はデータと同じように記憶装置に格納されておりデータとして処理することができるので，ビット構成を変更したり数を加えたりして，1つの命令を別の命令に変更することができる．自分自身を変更するような手続きを**純粋でない**（impure）**手続き**という．自分自身を修飾するようなプログラムを作るのは実はまずい方法である．他のプログラマにとってそれを解読することが極めて困難であるし，また，もっと悪いことには複数のプロセッサの間でその手続きを共有できない．純粋でない手続きを実行しているプロセッサはその内容を変更してしまうので，他のプロセッサが同じ手続きを実行しようとするときには元のと違った内容の命令やデータに変更されている．このような理由で純粋でない手続きは直ちに再使用することが不可能である．**純手続き**は自分自身を修飾しない．プログラムが使用されるとき命令が元のままであることを保証するために純手続き（**再入可能コード**）が用いられる．

1.2 プログラミング・システムの構成要素の発展

1.2.1 アセンブラ

プログラミング・システムの構成要素の発展を振り返ってみよう．

かつて，計算機のプログラマは，基本的な命令をハードウェアで解釈する原始的な計算機を自由に使用していた．プログラマは一連の1と0（機械語）によってこの計算機のプログラムを作り，機械の記憶装置に格納し，ボタンを押すと，計算機は動作を開始してこれらを命令として解釈した．

しかし，プログラマは機械語のプログラムを書いたり読んだりすることが困難であることがわかったので，より便利な言語を求めた．そして各機械の命令に対して**おぼえやすい記号名**（mnemonic：記号）を付けて，それを後で機械語に翻訳するようにして使用し始めた．このような記号名による機械語は今日**ア**

センブラ語と呼ばれている．アセンブラと呼ばれるプログラムが，自動的にアセンブラ語を機械語に翻訳するために開発された．アセンブラ・プログラムに対する入力は**原始プログラム**と呼ばれ，出力は翻訳された機械語（**目的プログラム**）である．

1.2.2 ロ ー ダ

アセンブラが目的プログラムを作り出すと，そのプログラムを記憶装置に入れて実行する必要がある．目的プログラムが実行形式として記憶装置に格納されるようにするのがローダの役割りである．

アセンブラは，目的プログラムを直接記憶装置に置き，それに制御を移して機械語のプログラムを実行させることもできる．しかしこの方法は，利用者のプログラムが実行されている間，記憶装置の中にアセンブラを入れ放しにするのでコア[1] の無駄使いである．また，プログラマは実行の度にプログラムを翻訳しなければならない．これは時間の浪費である．翻訳時間と記憶装置の浪費の問題を解決するために，システム・プログラマはローダと呼ばれるプログラムを開発した．

ローダは，プログラムを記憶装置に置き，実行のための準備を行なうプログラムである．単純な方法では，アセンブラはプログラムの機械語への翻訳結果を２次記憶装置に出力し，ローダがそれをコアに入れる．ローダは，記憶装置に利用者プログラムの機械語版を入れ，それに制御を移す．ローダ・プログラムはアセンブラよりはるかに小さいから，このようにすることにより利用者プログラムにもっと大きいコア領域を与えることができる．

多くの利用者が事実上同じプログラムを書いているようにするために“既製品”のプログラム（パッケージ）が開発された．これらのパッケージは計算機の製造者や利用者によって作られた．プログラマの技巧が上達するにつれて，自分のプログラムと既製品のプログラムを組合せ，結合することが望まれるようになった．この要求に応じて，利用者が幾つかの別のプログラムやサブルーチンを使用するようなプログラムを書く機能が用意されることになった．

---

1 従来の主記憶は磁気コアにより構成されている．したがって**記憶装置**と**コア**とを同義語として用いている．

サブルーチンとは，ある仕事を行なうため他のルーチンによって使用されることを目的として設計された一連の命令である．サブルーチンは閉じた型と開いた型の２つの型がある．**開いたサブルーチン**または**マクロ定義**とはこのコードが主プログラムの中に挿入される（実行の流れはそのまま続く）ものである．したがって，もし同じ開いたサブルーチンが４回呼出されると，呼出したプログラムの４個所でその内容が現われる．**閉じたサブルーチン**は主ルーチンの外部に格納できるものであり，制御がそのサブルーチンに移される．閉じたサブルーチンに関連して主プログラムは２つのこと，制御と移行とデータの転送，を行なわなければならない．

初期には，閉じたサブルーチンは記憶装置の定められた番地にロードする必要があった．例えば，利用者が平方根のサブルーチンを使用したいときは，プログラムが平方根のルーチン（SQRT）に割付けられた場所に制御を移すように主プログラムを作る必要があった．利用者のプログラムとサブルーチンは一緒にアセンブルされた．第２の利用者が同じサブルーチンを使用しようと思ったときには，やはり自分のプログラムと一緒にアセンブルした．このような固定のローディング方法によるコア割付けの例を **図 1.3** に示す．図ではコアは一次元の配列として示され，プログラムの領域は斜線で示されている．

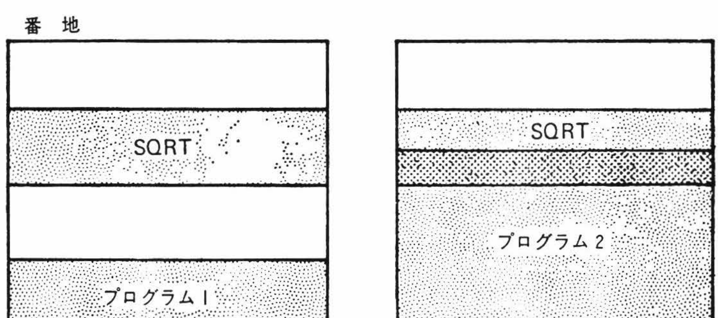


図 1.3　絶対型ローダのコア割付けの例

プログラム１はコアの内に“穴”があり，プログラム２では“**重なり** (overlay)”のために SQRT ルーチンの一部を壊してしまっている．

プログラマは互に記号名で参照できるようなサブルーチンを使用し，プログ

ラムの個々の部分の番地付けにわずらわされないようにすることを望んだ．すなわち，サブルーチンに記憶場所を割付け，記号名による参照に番地を代入するような計算機システムを望んだ．

システム・プログラムは，ローダがサブルーチンを利用者のプログラムの直後へ“再配置（relocate）”できるような形に翻訳できれば，一層効果的であると考えた．プログラムをコアの任意の場所に置いて実行できるように調整することを**再配置**（relocation）という．**再配置型**ローダは次の4つの機能を持つ：

1. プログラムに記憶装置の領域を割付ける（**割付け**：allocate）
2. 目的プログラムの間の記号名参照を解決する（**結合**：linking）
3. 割付けられた領域に関する番地定数などの，番地に関係する場所を調整する（**再配置**：relocation）
4. 機械語とデータを物理的に記憶装置に入れる（**ローディング**：loading）

これから論じようとする種々のローダ（コンパイル・実行型，絶対型，再配置型，直接結合型，ダイナミック・ローディング型およびダイナミック・リンキング型）は主としてこれら4つの基本的な機能の実行方法に相違がある．

利用者プログラムが実行している期間を**実行時**（execution time）という．利用者の原始プログラムが翻訳されている期間を**アセンブル時**または**コンパイル時**という．**ロード時**とは実行するために目的プログラムをロードしたり用意したりする期間をいう．

1.2.3 マ　ク　ロ

プログラマがプログラムの中で全く同一の部分を繰り返して書かなくてもすむようにするために，オペレーティング・システムはマクロ処理機能を持っている．マクロ機能を用いることによってプログラマは，プログラムのある部分に対して省略形を定義し，プログラム中でこの省略形を使用することができる．マクロ・プロセッサは省略形によって定義されたプログラムの部分を**マクロ定義**として処理し，この定義を保存する．マクロ・プロセッサは，プログラムの中でこの省略形が出てくる（マクロ呼出し）と，その定義を代入する．

プログラマがプログラムを省略形で書けるようにするほかに，マクロ機能は，

一般的なテキスト処理プログラムとして，また，個々の計算機設備に対するオペレーティング・システムの特別化のために使用されている．オペレーティング・システムの特別化(システム編集：system generation）のために，すべてのオペレーティング・システムは一連のマクロ定義として書かれている．オペレーティング・システムを特別化するためには，一連のマクロ呼出しが書かれる．これらのマクロ定義とマクロ呼出しはマクロ・プロセッサによって処理され，適当な定義を代入して，オペレーティング・システムのすべてのプログラムを編集する．

1.2.4 コンパイラ

利用者の問題が科学，事務および統計の計算というように細分化するにつれ，問題向きの言語（**高水準言語**：high level language）が開発され，利用者が問題を簡潔に容易に表現できるようになった．FORTRAN, COBOL, ALGOL および PL/I といった高水準言語はコンパイラやインタプリタによって処理される．**コンパイラ**とは高水準言語で書かれたプログラムを入力とし，目的プログラムを出力するプログラムである．**インタプリタ**とは原始プログラムがあたかも機械語であるかのように解釈実行するプログラムである．同一の名前(FORTRAN, COBOL など）は，しばしば，コンパイラとそれに対応した言語との両方を意味して使用される．

新しいコンパイラは，今日プログラマが要求している複雑な機能をかなえるものでなければならない．コンパイラは，PL/I, COBOL および ALGOL 68 のような言語で使用されるポインタ変数やデータ構造に対する複雑な参照手段を備えていなければならない．新しいコンパイラは，計算機のハードウェア割込みに関する演算文（例えば，PL/I の条件文など）を処理するために，オペレーティング・システムと密接な関係を持たなければならない．

1.2.5 形式的体系

形式的体系とは解釈のない算法である．これはアルファベット，公理と呼ばれる語の集合，および，導出の法則と呼ばれる有限個の関係の集合とで構成されている．形式的体系の例は，集合論，ブール代数，Post の体系および Ba-

ckus Normal Form があげられる．形式的体系はプログラム言語の設計，作成および研究に重要になりつつある．特に，これらはプログラム言語の**構文**(syntax：形式）と**意味論**を記述するのに使用できる．これらは直構文コンパイル，コンパイラの証明および言語の複雑さの研究に使用されている．

1.3 オペレーティング・システムの発展

たった数年前まで FORTRAN プログラマは，計算機を使用するとき，左手に自分の原始デックを，右手に FORTRAN コンパイラなる緑色のデックをかかえてやってきた．そして，

1. FORTRAN コンパイラ(緑色のデック)をカード・ホッパに置きロード・ボタンを押す．計算機は FORTRAN コンパイラをロードする．
2. 原始デックをカード・ホッパに置く． FORTRAN コンパイラはこれを機械語のデックに翻訳して赤色のカードにせん孔出力する．
3. “ローダ”と書かれた桃色のカード・デックをカード・ライブラリから取り出す．計算機はローダを記憶装置にロードする．
4. 新しく翻訳された目的デックをカード・ホッパに置く．ローダはこれを機械にロードする．
5. プログラムが呼出すサブルーチンのデックをカード・ホッパに置く．ローダはこれらのサブルーチンをロードする．
6. 最後に，ローダは利用者のプログラムに制御を移す．利用者のプログラムはデータ・カードの読込みを行なうかも知れない．

この多色デック・システムは幾つかの点で不満足であり，より柔軟なシステムに対する強い要求があった．1つの理由は，ジョブ（**ジョブ**とは計算機の仕事の単位である．例えば，プログラムのアセンブルなど）とジョブの合間でカードを取り扱う間機械が何ら仕事をせず，貴重な計算機時間が浪費されたことである．この浪費をなくするには,幾つものジョブをカード・ホッパに積んで読込むという**バッチ・ジョブ**の機能が用意された．**バッチ・オペレーティング・システム**はジョブを“束”にして実行する．例えば，バッチ・システムは上述のステップ1から6を， FORTRAN コンパイラやローダを2次記憶装置から

取り出して実行する.

計算機時間や記憶装置，入出力装置およびファイルに対する要求が増大するにつれて，これら資源の効率良い管理が一層重要となってきた．第9章でこれらの管理の種々の方法を論じる．これらの資源は貴重であり，能率の悪い管理は経費の増大を招く．各資源の管理は，これらを使用する経費と複雑さが増大するにつれて発展してきた.

単純なバッチ・システムでは，全部の記憶装置は1つのプログラムに割当てられた．したがって，プログラムが全部の記憶装置を必要としないとき，資源の一部は浪費されていた．**コア記憶装置を分割**する多重プログラミングのオペレーティング・システムはこの問題を解決するために開発された．**多重プログラミング**とは同時に多数のプログラムを分割したコアの領域に入れる方法である．プログラムはコアの固定長の領域を与えられる場合（**固定タスク数の多重プログラム：MFT**）と可変長のコアの領域を与えられる場合（**可変タスク数の多重プログラム：MVT**）とがある.

このような記憶装置の分割を行なうシステムでは，しばしば，ある領域が1つのプログラムを入れるのには小さすぎるために，使用できないことがある．この"穴"，すなわち，コアの使用されない領域の問題は**断片**（fragmentation）と呼ばれる．断片は再配置の技法によって(Burroughs 6500)，または，ページングによって(XDS 940, HIS 645)小さくすることができる．**再配置型コアの分割**によって未使用領域を1つに集め，連続したコアの領域とすることができる.

ページングはプログラムが等長の部分――**ページ**――に分割され，またコアも等長の領域――**ブロック**――に分割され，ページがブロックにロードされるような記憶装置割付けの方法である.

ページングの技法には単純型とデマンド型の2つがある．**単純ページング**では実行のためのプログラムのすべてのページがコアに入っていなければならない．**デマンド・ページング**ではすべてのページがコアに入ってなくてもプログラムを実行することができる．すなわち，ページは必要になったとき（デマンド）コアの中に持ってこられる.

読者は第1.1節で，複数のプロセッサを持っているシステムは多重処理シス

テムと名付けられたことを思い出してほしい．**トラフィック制御** (traffic controller) はプロセッサとプロセスの仲介をする．プロセッサの時間という資源は**スケジューラ**と呼ばれるプログラムによって割当てられる．I/O に関係するプロセッサは I/O プロセッサといわれ，このプロセッサのプログラムを作ることを **I/O プログラミング**という．

情報のファイルの資源は**ファイル・システム**によって割当てられる．**セグメント**とは利用者が 1 つの物として取り扱いたい情報のかたまりである．**ファイル**はセグメントである．ファイルには (1) ディレクトリーと (2) データまたはプログラムの 2 種がある．**ディレクトリー**は他のファイルの位置の情報を納めている．階層的なファイル・システムでは，ディレクトリーは他のディレクトリーを指すことができ，そのディレクトリーは再び他のディレクトリーまたはファイルを指すことができる．

タイム・シェアリングとはプロセッサ時間の割当て方式の 1 つである．この方式は各利用者に迅速な応答を行なうための会話型処理と CPU 時間のタイム・スライスによって特徴付けられる．

仮想記憶 (virtual memory : **名前空間，番地空間**) は計算の実行の間にプロセッサによって作られる番地から成る．**記憶空間**は物理的な記憶装置の場所に対応する番地の集合から成る．**セグメンテーション**の技法は大きな名前空間と優れた保護機構を実現する．保護と共有はセグメントの制御された参照を許す手段である．

1.4 利用者から見たオペレーティング・システムの機能

利用者から見たとき，オペレーティング・システム（モニタ）は問題を解く専門家として援助をするものである．特に次の機能がシステムによって遂行される：

1. 仕事の順序付け，スケジューリングおよび交通整理の働き．

2. 入力／出力プログラミング．

3. 利用者から自分自身を保護すること．利用者を他の利用者から守ること．

4. 2次記憶装置の管理.

5. 誤り処理.

1人の利用者が4時間かかる仕事を持っており，他の利用者が4秒で終る仕事を持っているとする．両方の仕事が同時に計算機に加えられたとき，4秒の利用者の仕事を先に実行することがより適切であろう．このような考え方に基づいて，仕事のスケジューリングはオペレーティング・システムによって自動的に遂行される．多くの計算機システムのように，入力および出力がプログラム実行と同時に実行されるとき，これらすべての機能はトラフィック制御によってスケジュールされる.

既に述べたように，I/O チャネルは，特別の命令を持つ独立の計算機と考えることができる．たいていの利用者はチャネル・プログラムの作り方を学びたいとは思わない（多くの場合非常にこみいった仕事である).利用者はプログラムの内で“読込め”とだけいって，モニタ・システムに，I/O チャネルで実行するプログラムを用意させる．このような機能はオペレーティング・システムに用意されている．多くの場合 I/O チャネルに与えられるプログラムは次のように処理を行なう密接に関連した一連の割込みルーチンから成っている：

“I/O チャネル君，文字を受信しましたか？”

“はい，受信しましたよ.”

“確かですか？”

“はい.”

“よろしい．それでは次を送りますよ.”

“はい，どうぞ.”

“次の文字を送ってよいのですね？”

“送りなさい”.

オペレーティング・システムの特に重要な機能は，利用者が他の利用者から，故意にまたは事故により，傷つけられるのを保護することである．すなわち，他の利用者が勝手にプログラム，ファイルおよびデータ・ベースを実行したり変更したりしないように保護することである．オペレーティング・システムは不可侵を保証しなければならない．利用者を他の利用者から守ると同時に，オペレーティング・システムは，自分自身を故意もしくは事故で“ぶち壊す”利

用者から保護しなければならない．

学生達は保護機構に対する強力な挑戦者である．MIT においてシステム・プログラミングの科目を選択する学生が非常に多いので，機械語で実行されるプログラムのすべてを1人ずつ採点することが困難である．そこで単純な問題に対しては，オペレーティング・システムの一部として含まれる採点プログラムを作った．確か第1問は1つのレジスタの中のAの個数を数えて答えを他のレジスタに入れるというものであったが，採点プログラムは学生のプログラムを呼出し，制御を移す．この単純な問題においては学生のプログラムはレジスタの内容を処理して他のレジスタに答えを入れ，採点プログラムに復帰する．採点プログラムは答えのレジスタに正しい数が残されたかどうかを判定し，後でクラスのすべての学生の名前と得点の一覧表を印刷する．例えば，

VITA KOHN	—	CORRECT
RACHEL BUXBAUM	—	CORRECT
JOE LEVIN	—	INCORRECT
LOFTI ZADEH	—	CORRECT

昨年は，計算機の表は次のように始まっていた：

JAMES ARCHER	—	CORRECT
ED MCCARTHY	—	CORRECT
ELLEN NANGLE	—	INCORRECT
JOHN SCHWARTZ	—	MAYBE

(John Schwartz はどのようにして問題を解いたか知らないが，彼にAを付けた．)

2次記憶装置の管理は，利用者のプログラムやデータに対するディスク，テープおよびその他の2次記憶装置の利用に関して，オペレーティング・システムによって行なわれる．

オペレーティング・システムは誤りの処置をしなければならない．例えば，プログラマがレジスタをあふれさせたとき単に計算機を停止させてオペレタが介入するのを待つというのは経済的ではない．誤りが発生したとき，オペレーティング・システムはあらかじめ定めておいた処置をとる．

1.5 利用者から見たオペレーティング・システムのバッチ制御言語

多くの利用者はオペレーティング・システムをバッチ・システムの制御カードを通じてのみ見ている．この制御カードはプログラムの先頭に置かなければならない．本節では単純なモニタ・システムとそれに関連した制御カードについて論じる．もっと複雑なモニタについては第9章に述べている．

モニタとはオペレーティング・システムの制御プログラムを指す言葉である．バッチ・システムでは，ジョブはカード読取り機に積まれ，モニタ・システムは順番に各ジョブを処理する．1つのジョブは順番に実行される幾つかの独立のプログラムから成っている．各々のプログラムは**ジョブ・ステップ**と呼ばれる．**バッチ・モニタ・システム**の場合，利用者は制御言語を用いて実行したい仕事をシステムに通知する．単純なバッチ・モニタ・システムにおいては2種の制御カード，実行カードと定義カード，を有している．例えば，実行カードは次の形式である：

//ステップ名　EXEC　　実行すべきプログラム名，引数1，引数2

定義カードの1つであるジョブ制御カードは次の形式をとる：

//ジョブ名　JOB　　（利用者名，課題番号，予想時間，印刷行数，穿孔枚数）

通常書式 /* のファイル終了カードが，データ集合の終了を示す．FORTRAN ジョブの例を考える．

```
//EXAMPLE     JOB      DONOVAN,     T168,1,100,0
//STEP1       EXEC     FORTRAN, NOPUNCH
           READ 9100,N
           DO 100 I = 1,N
           I2 = I*I
           I3 = I*I*I
       100 PRINT 9100,I,I2,I3
      9100 FORMAT (3I10)
           END
//STEP2       EXEC LOAD
//STEP3       EXEC OBJECT
```

最初の制御カードは定義カードの例である．利用者は Donovan であると定義する．システムは利用者のための会計ファイルを開設し，これに予想の処理

時間は1分，出力行数は100行，カードのせん孔は行なわないことを記入する．次の制御カード，EXEC FORTRAN,NOPUNCH は実行カードの例である．これは，システムがプログラム FORTRAN を実行させ，1つの引数 NOPUNCH を与えることを示す．この引数はモニタ・システムの効率をより良くするものである．すなわち，カードはせん孔されないから，せん孔ルーチンを利用する必要はない．コンパイラに対するデータは例に示された FORTRAN プログラムであり，ファイル終了カード /* によって終結している．

次の制御カードは実行カードの別の例であり，この場合はローダを実行させる．ちょうど今コンパイルされたプログラムが，実行のために必要なすべてのルーチンと共にロードされる．ローダはサブルーチンを主プログラムに"結合"する．このジョブ・ステップはファイル終了カードによって終結する．EXEC OBJECT カードは他の実行カードであり，モニタ・システムは今コンパイル

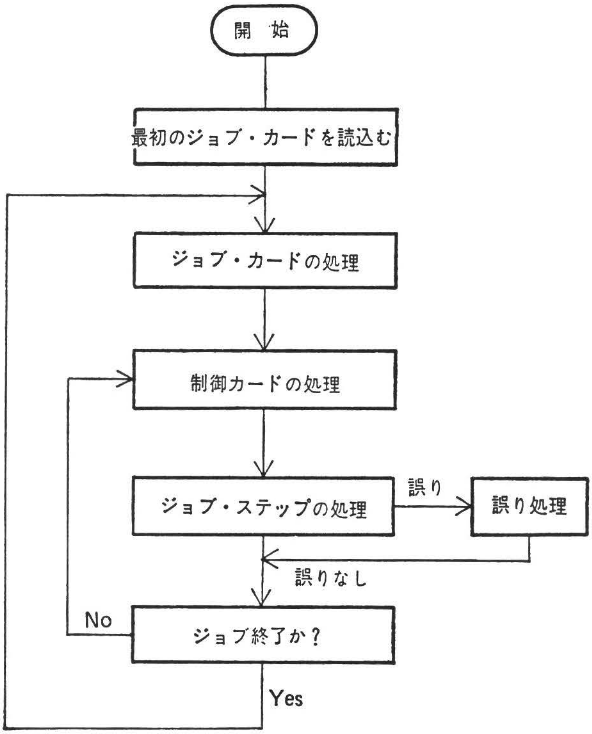


図 1.4 簡単なバッチ・モニタ・システムの主ループ

され結合された目的プログラムを実行させる．データ・カード 10 はプログラムへの入力でありファイル終了カードが続いている．

図 1.4 に示した単純なループはバッチ・モニタ・システムの組込みの概略を示している．モニタ・システムは最初のカード，それは多分ジョブ・カードである，を読込まなければならない．ジョブ・カードを処理するとき，モニタは利用者名，会計番号，割当て時間，カードせん孔の最大枚数および印刷行数の最大数を保存する．次の制御カードが実行カードであれば，モニタは2次記憶から対応したプログラムをロードし，実行プログラムに制御を移してジョブ・ステップの処理を行なう．処理の途中に誤りが発生すれば，システムは誤りを報告し次のジョブ・ステップの処理に移る．

1.6 利用者から見たオペレーティング・システムの設備

アプリケーションを主とする利用者にとっては，オペレーティング・システムの機能とは問題を解くのに手助けとなる設備を用意することである．スケジューリングや保護の問題には興味がなく，どのようなソフトウェアが使用できるかが問題である．次の設備が新しいオペレーティング・システムに用意されている：

1. アセンブラ
2. FORTRAN, COBOL および PL/I などのコンパイラ
3. SINE, COSINE, SQUARE ROOT などのサブルーチン・ライブラリ
4. サブルーチンを結合し，プログラム実行の準備をするリンケージ・エディタやプログラム・ローダ
5. SORT/MERGE や TAPE COPY などのユーティリティ・ルーチン
6. 回路解析やシミュレーションなどの応用パッケージ
7. プログラムのトレーシングや“コア・ダンプ”などのデバッグ機能
8. データ管理とファイル処理
9. システムのハードウェア管理

これらの“設備”の面からオペレーティング・システムを見ることは利用者にとって大変興味のあることであるけれども，“オペレーティング・システム

はコンパイラを幾つ持っているか”という問に対する答えは，オペレーティング・システムの構造と効率に関してよりも，各計算機会社の市場での力関係の方をより多く語っているように思われる．

1.7 ま と め

プログラミング・システムの主要な成分は，

1. アセンブラ

アセンブラへの入力は**アセンブラ語のプログラム**である．出力は目的プログラムと，ローダによって目的プログラムの実行の準備ができるようにするための情報である．

2. マクロ・プロセッサ

マクロ呼出しとはあるコードに対する省略形（すなわち名前）である．**マクロ定義**は（マクロ呼出しの）名前を持った一連のコードである．**マクロ・プロセッサ**はマクロ呼出しに対してマクロの定義を代入し，特別仕立てをするプログラムである．

3. ロ ー ダ

ローダとは目的プログラムをロードし，実行の準備をするルーチンをいう．ローディングの方法には絶対型，再配置型および直接結合型などがある．一般にローダは目的プログラムを**ロード**し，**再配置**し，**結合**しなければならない．

4. コンパイラ

コンパイラは“高水準言語で書かれた”原始プログラムを入力とし，対応した目的プログラムを出力するプログラムである．

5. オペレーティング・システム

オペレーティング・システムは記憶装置，プロセッサ，入出力装置および情

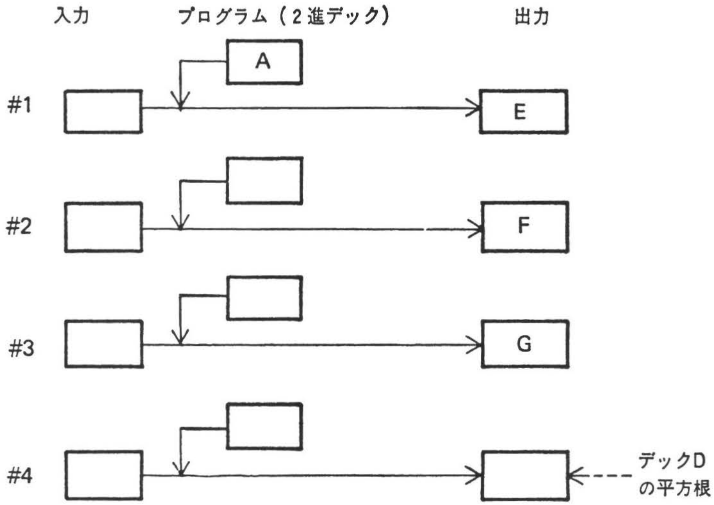
報などの資源やサービスの割当てを行なう．オペレーティング・システムは，**トラフィック制御**，**スケジューラ**，**記憶装置管理モジュール**，**I/O プログラム**および**ファイル・システム**などの資源を管理するプログラムを持っている．

演　習　問　題

1. 次の違いを述べよ：
   プロセッサと手続き（procedure）
   手続きとプログラム
   プロセッサとI/Oチャネル
   多重処理と多重プログラミング
   開いたサブルーチンと閉じたサブルーチン
2. 記憶装置のビットはデータまたは命令を表わす．手続きは，特定の場所に与えられたものが命令であるかデータであるかをどのようにして見分けるか．
3. IBM 360 型の計算機があり，次の入力デックが与えられたとする：
   デックA：**2進コード**（機械語）で書かれた基本アセンブラ語（BAL）のアセンブラ
   デックB：BAL で書かれた FORTRAN から BAL への翻訳ルーチン
   デックC：データ・カードを読込みそのデータの平方根を印刷する FORTRAN プログラム
   デックD：デックCの平方根プログラムに対するデータ・デック

   平方根を計算するためには4回の計算機処理を行なわねばならない．この実行は下の図のように（左から右へと）1つのプログラムによって入力デックが読込まれ，出力デックが作られる．もちろん，実行の結果作られた出力デックは，以下の実行において入力デックまたはプログラムとして使用してよい．

   次の図において名前の付いていないデックにA，B，C，D，E，F，Gを記入せよ．



4. ハードウェア（止めねじ，ねじ，トランジスタなどからできている物理的な装置）とソフトウェア（カード，テープ，ディスクなどに記録された2進数の情報で，記憶装置にロードしてプログラムやデータとして使用されるもの）の区別がある．次にあげたものはいずれか：

a. コンパイラ

b. プロセッサ

c. オペレーティング・システム

d. ローダ

e. I/O チャネル

f. コア記憶装置

g. アセンブラ

h. ファイル

i. モニタ

j. ディスク駆動装置

5. 第1.5節で単純なバッチ・モニタを論じた．この間ではもっと機能を与え，その構造を考えてみる．資源管理に対して利用者により多くの制御能力を与えるために**データ・セット**の概念を導入する．データ・セットとはデータの源または倉庫であり，物理的にはディスク，カード読取り機などの形態をと

る．利用者はデータ・セットを新しい制御カードによって定義する：

//データセットの論理名　DD　パラメータ

例として第1.5節の FORTRAN コンパイラの場合の SYSPRINT は原始言語の製表を行なうデータ・セットに対する論理名であり，

//SYSPRINT DD UNIT=00E

は製表が番号 00E（印刷機）の装置に行なわれるべきことを示している．コンパイラは上のほかに次のデータ・セットを使用する：

SYSIN	（原始カード入力；通常カード読取り機）
SYSLIN	（目的コード出力；通常ディスク）

ローダは次のデータ・セットを使用する；

SYSLIN	（翻訳プログラムの出力）
SYSPRINT	（メッセージ；通常印刷機）
SYSLIB	（ライブラリ・サブルーチン；通常ディスク）

利用者プログラムは次のデータ・セットを使用する：

SYSIN	（データ；通常カード読取り機）
SYSPRINT	（印刷機；通常印刷機）

データ・カードを制御カードと同じデックに入れるには，＊印をＤＤカードのパラメータ部に記入し，“次は入力流れである”ことを示す．データ・カードは /*（データ・セットの終了）カードで終了していなければならない．また DD * データ・セットはジョブ・ステップの最後のデータ・セットでなければならない．

番号 141 のディスクと番号 00E の印刷機を持ったシステムで，上のすべてのデータ・セットを持つ第1.5節のジョブは次のように定義される．

モニタは利用者の制御カードを読込み，要求された資源を割当て，要求さ

```
// EXAMPLE        JOB       DONOVAN, T168, 1, 100, 0
// STEP1          EXEC      FORTRAN, NO PUNCH
// SYSPRINT       DD        UNIT = 00E
// SYSLIN         DD        UNIT = 141
// SYSIN          DD        *
            READ        9100,N
            DO    100   I=1,N
            I2 = I*I
            I3 = I*I*I
   100      PRINT       9100,I,I2,I3
   9100     FORMAT      (3I10)
            END
/*
// STEP2          EXEC      LOADER
// SYSPRINT       DD        UNIT = 00E
// SYSLIB         DD        DSNAME = FORTLIB
// SYSLIN         DD        FROM STEP1.SYSLIN
// STEP3          EXEC      OBJECT
// SYSPRINT       DD        UNIT = 00E
// SYSIN          DD        *
            10
```

れたジョブ・ステップの実行を開始しなければならない.

モニタの仕事を次の2つの部分に分けたとしよう：

1) 読込みルーチン／解釈ルーチン（RDR/INT）は制御カードを読込み，解釈し，制御カードの情報からデータ・ベースを作る.

2) 開始ルーチン／終了ルーチン（INIT/TERM）は RDR/INT によって作られたデータ・ベースを用いて，資源割当てのスケジュールを行ないジョブ・ステップを開始し，終了する.

a. 利用者が選んだ物理的な装置に論理名を付けられることの意義は何か.

b. RDR/INT は，INIT/TERM が DD 制御カードの機能を有効にするためにどのようなデータ・ベースを作らねばならないか.

c. ほかにどのようなデータ・ベースが必要か.

d. RDR/INT が順番にカードを読込むとき，どの時点で制御を INIT/TERM に移してジョブ・ステップを実行するか.（DD * のデータ・セットに含まれるデータ・カードは RDR/INT で読込まれるのでは

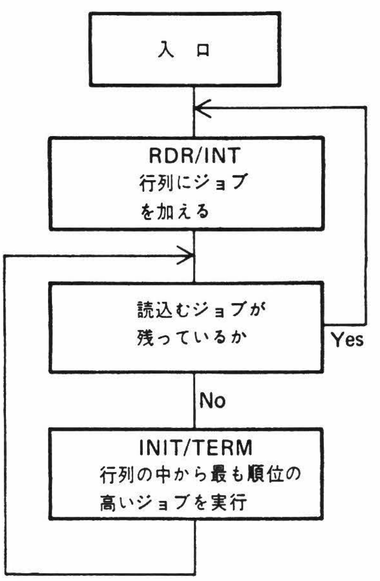
なく，実行しているプログラムによって読込まれることに注意.)

e. 各ジョブ・ステップ終了時に RDR/INT の各データ・ベースに何が起こるか.

f. 各ジョブの終了時に各データ・ベースに何が起こるか.

g. どのような誤りが RDR/INT によって検出されるか. INIT/TERM ではどうか. どのような処理が行なえるか.

6. バッチ・モニタの機能をもっと大きくするために，JOB カードに PRIORITY（優先順位）と呼ばれる新しいパラメータを追加したい. 優先順位は0から9の数で，大きい数のジョブが先に実行される. もちろん料金も高い. 明らかに一時に1つのジョブのカードだけを読込むということはできなくなるので**待ち行列**（queue）の機能を追加しなければならない. RDR/INT によってジョブ毎に作られたデータ・ベースは，読込むべきジョブがなくなるまで待ち行列に入れられ， INIT/TERM は最も優先順位の高いジョブを実行する. 以後新しいジョブが待ち行列に加えられる必要のあるときは加え，それらの中で最も優先順位の高いジョブが実行される：



a. RDR/INT により作成されるデータ・ベースに何を追加しなければな

らないか.

b. 第1.5節のモデルにおいてDD * データ・セットの中のデータ・カードは省略した. このデータ・セットがあればどうなるか.

c. PRIORITY パラメータの意義は何か.

第 2 章

計算機の構造, 機械語およびアセンブラ語

本章の目的は計算機の構造, 機械語およびアセンブラ語を述べることである.

例として IBM システム/360 と 370[1] を用いる. ここで目的とするのは特定のアセンブラ語を教えようとすることではなく, アセンブラの設計 (そして後にはコンパイラの設計) を説明するのに必要な内容に限定して述べている. ここで行なっている IBM 370 のアセンブラ語の紹介は他の参考文献により補足する必要がある. (第10章参考文献——計算機の構造を参照.)

本章は2組の人達, アセンブラ言語をよく知っており IBM 370 について知りたい人, および, アセンブラ語でプログラムを書いたことのない人で計算機のマニュアルを読むための手引き書としようとする人を対象として書いてある. 説明と例題は容易に他の機械に移し変えることができる.

2.1 一般的な計算機の構造

今日の計算機のほとんどすべては, 数学者 John Neumann の考案した"プログラム記憶型計算機"の概念に基づいている. 図 2.1 は IBM システム /360 などの典型的な von Neumann 型計算機の構造を示している.

CPU は命令解釈部, ロケーション・カウンタ, 命令レジスタおよび各種の作業用レジスタと汎用レジスタから構成されている. **命令解釈部**は一群の電気

1 IBMシステム /360 (または単に 360) は1964年以降 IBM で生産されている計算機のシリーズの名前である. これらのすべては互換性のある命令とマニュアルを持っている. 360 の改良版である IBM 370は1970年から生産されている. 370は360と互換性がある.

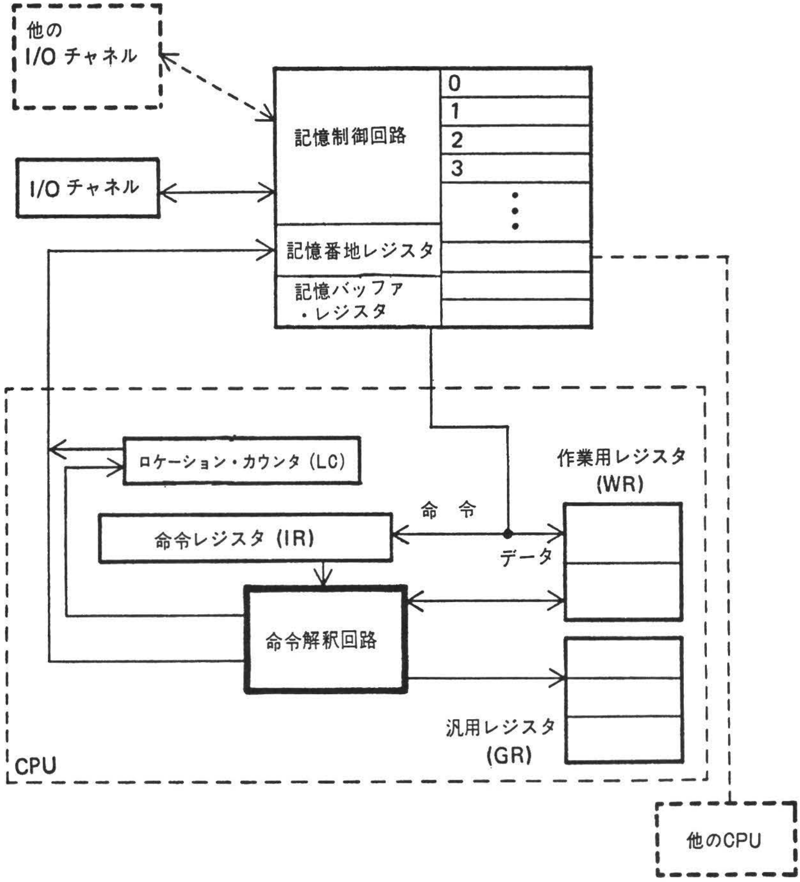


図 2.1 一般的な計算機の構造

回路（**ハードウェア**）から成り，記憶装置から取り出された命令を解釈して実行する．**ロケーション・カウンタ**（LC）は**プログラム・カウンタ**（PC）や**命令カウンタ**（IC）とも呼ばれ，実行される命令の場所を示すためのハードウェアの記憶装置である．現在の命令の写しは**命令レジスタ**（IR）に納格されている．**作業用レジスタ**は命令解釈部のための“走り書きのメモ（scratch pad)”の役を果す記憶装置であり，一方**汎用レジスタ**はプログラマによって記憶装置として，または特別の機能を実行するための装置として使用される．

記憶装置と CPU の間の主要なインタフェースは記憶番地レジスタと記憶バ

ッファ・レジスタである．**記憶番地レジスタ**（MAR）は読出したり書込んだりする記憶場所の番地を格納している．**記憶バッファ・レジスタ**（MBR）は“読出し”の後では MAR によって指定された記憶場所の写しを，“書出し”に先立っては記憶場所の新しい内容を納めている．**記憶制御装置**は MBR と，MAR が指定する記憶場所との間でデータを転送するハードウェアである．

I/O チャネルは記憶装置から情報を入力したり出力したりするために特別の命令を解釈する独立の計算機と考えることができる．

これらの機械の構成要素がどのように関連して働くかを説明するために簡単な計算機を考える．この計算機は2進数 00，01，10および11で示される4つの汎用レジスタを持つ．基本的な命令形式は次に示すとおりである：

命令コード	レジスタ番号	記憶番地
(op)	(reg)	(addr)

例えば，命令

ADD　　2, 176

は記憶場所 176 に格納されたデータが，汎用レジスタ 2 の内容に加算される．その結果はレジスタ 2 に残される．**図 2.2** のマイクロ流れ図は，命令を実行する命令解釈部の中で実行される一連のハードウェア動作を説明している．

細部に関しては計算機によって異なるが，この基本的な機械構成はすべての通常の計算機に当てはまる．

2.1.1　新しい計算機を知るための一般的な方法

本節で概説するのは新しい計算機を知るためにとられる方法である．それは新しい計算機のプログラムを作ろうとするときに持つ一連の質問に対する答えを見出すことである．

まず，これらの質問をあげ，次に IBM 360 および 370 に対してこれらの質問に答えよう．第 9.1 節では（独立の計算機と考えることのできる）I/O チャネルに関してこれらと同じ質問を行なう．

1.　記憶装置

記憶装置の基本的な単位，容量および番地指定法は何か？

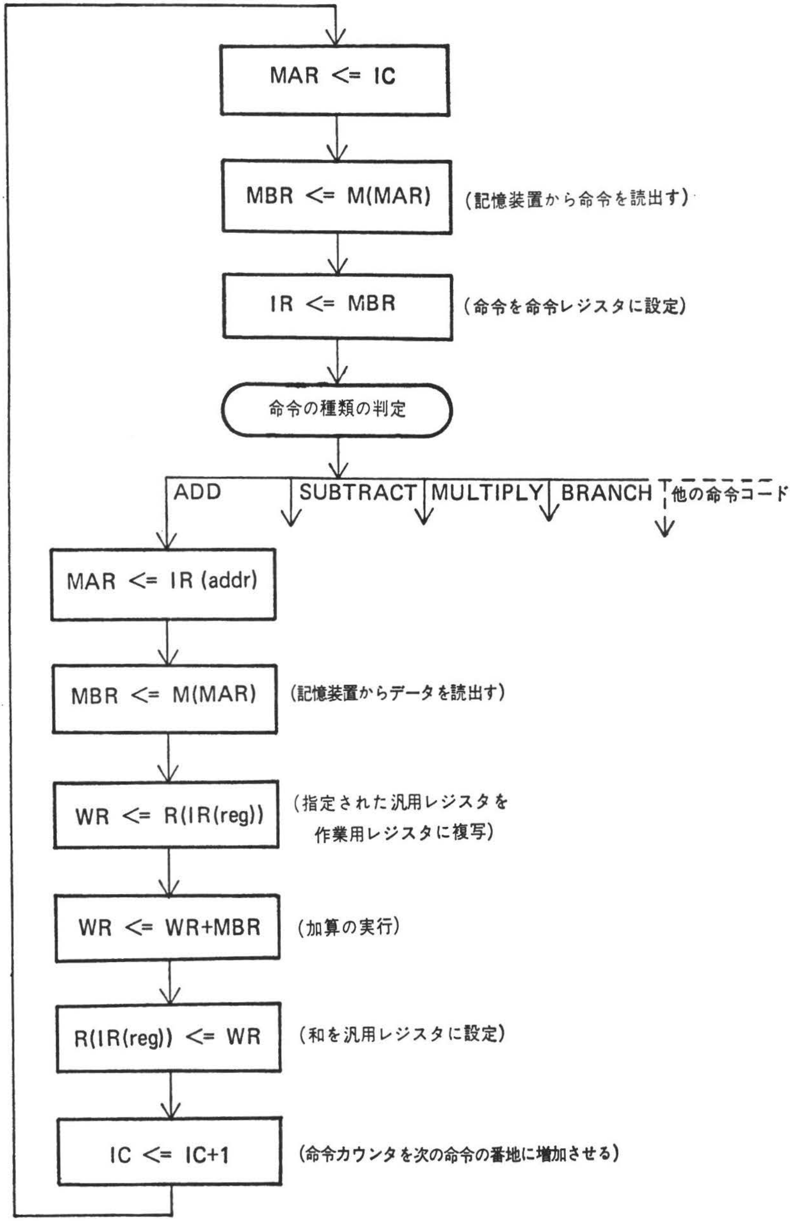


図2.2 ADD 命令に対するマイクロ流れ図の例

2. レジスタ

レジスタが何個あるか？　容量，機能および相互の関係はどうか？

3. データ

どの型のデータが計算機で処理できるか？　文字，数，論理データは処理できるか？　データはどのようにして格納されるか？

4. 命　令

機械命令にはどのような種類があるか？　算術演算命令，論理演算命令，記号処理命令はあるか？　形式はどうか？　どのように記憶装置に格納されるか？

5. 特殊機能

機械の割込みの構造はどうか？　利用者にはどのような保護機構が使用できるか？

2.1.2　計算機の構造——IBM 360と370

本節では IBM 360 について前述の質問に答えよう．内容は IBM 370 に対しても等しく適用できる．

1.　記憶装置

IBM 360 の記憶装置の基本単位はバイト，8ビットの情報，である．すなわち，記憶装置の番地付けのできる各々の位置は8ビットの情報を納め得る．また，連続したバイトを基本単位として処理する機能がある．これらの基本単位は次のとおりである：

記憶装置の単位	バイト数	ビット数
バイト	1	8
半語	2	16
語	4	32
2語長語	8	64

時には4ビットから成る記憶装置の単位を**ニブル**（nibble）ということがある．

IBM 360の記憶装置の容量は最大 2^{24} バイト（約1,600万）である．

IBM 360の記憶装置の番地指定は3つの成分から構成することができる．番地の値はオフセットの値にベース・レジスタの内容を加え，インデックス・レジスタの内容を加えたものに等しい．この番地指定の例を後に示すことにする．

一般に記憶装置の単位に対する操作は番地の**値の小さい**バイトによって指定される．例えば，語（4バイト）の番地を指定するには番地の値の小さいバイトで語の番地とする．

2.　レジスタ

IBM 360は各々32ビットから成る16個の汎用レジスタを持つ．加えて各々64ビットから成る4つの浮動小数点レジスタがある．また，ロケーション・カウンタの値，保護情報および割込み状態を含む64ビットのプログラム状態語（psw）を持つ．

汎用レジスタは各種の算術演算や論理演算のため，およびベース・レジスタとして使用することができる．プログラマがこれを算術演算または論理演算で使用するとき，これらのレジスタは，数を加えたり，減じたり，比較したりするための走り書きのメモ（スクラッチ・パッド）と考えられる．ベース・レジスタとして使用されるときは，これらは番地の形成に関与する．例として次の命令を考える．

```
                インデックス・レジスタ
                        ↓
A            1,901 (2, 15)
                ↗        ↗
            オフセット    ベース・レジスタ
```

これは加算命令と解釈される．1つの数がレジスタ1の内容に加算される．

数の記憶場所は901（**オフセット**）＋レジスタ2（**インデックス**）の内容＋レジスタ15の内容（**ベース**）である．すなわち，これら3つの数が加え合わされて，その結果がレジスタ1の内容に加えたい数を格納している記憶場所の番地とされる．

番地の形成においてなぜこんなに複雑なことが必要なのかと思うであろう．その動機は2つある．第1，ベース・レジスタはプログラムの再配置処理を容易にすることである．後でわかるように，プログラム全体は1つの場所から他

へとベース・レジスタ[2] の内容を変更することによって移動することができる．しかしながら，ベース・レジスタを用いることの主たる動機は，コアの効果的な番地指定を可能にすることにある．例えば，IBM 360 のすべてのコアの場所（1,600万）をベース・レジスタを用いずに番地指定するためには番地部が24ビット必要である．前述の加算命令が次図に示したようにコアの中で作られるとすると，それを格納するために命令コードに8ビット，数が加算される16個の内の1個のレジスタを指定するために4ビット，16個の内1個のインデックス・レジスタを指定するために4ビット，そして最後に加えたい数の番地を指定するために24ビット，合計40ビットが必要である．

命令コード	引数レジスタ	インデックス	番　地	
8ビット	4ビット	4ビット	24ビット	計：40ビット

ベース・レジスタを使用すると，命令を次の形式で格納することができる．4ビットを使用して16個の内の1個のレジスタをベース・レジスタとして指定し，オフセットとして12ビットを使用すると，加算命令については合計32ビットでよく，番地の指定について8ビットの節約となる．

命令コード	引数レジスタ	インデックス	ベース	オフセット	
8ビット	4ビット	4ビット	4ビット	12ビット	計：32ビット

この短い形式の欠点は，実行時の番地形成に対するオーバヘッドと，12ビットの長さのオフセットは0から4,095までの数しか指定できないという点である．したがって，データに“到達する”ことが難しい．すなわち，番地の指定をしたいコアの場所はベース・レジスタの指しているコアの場所から4,096以上離れてはならない．

3. データ

IBM 360 は図 **2.3** に示したような幾種かの書式のデータを格納することが

2　ベース・レジスタは再配置の問題を完全に解決するわけではない．難かしい番地定数の問題も解決しなければならない．**番地定数**とは，これを用いてプログラマが特定の記憶場所の番地の値を記憶することができるようにするものである．

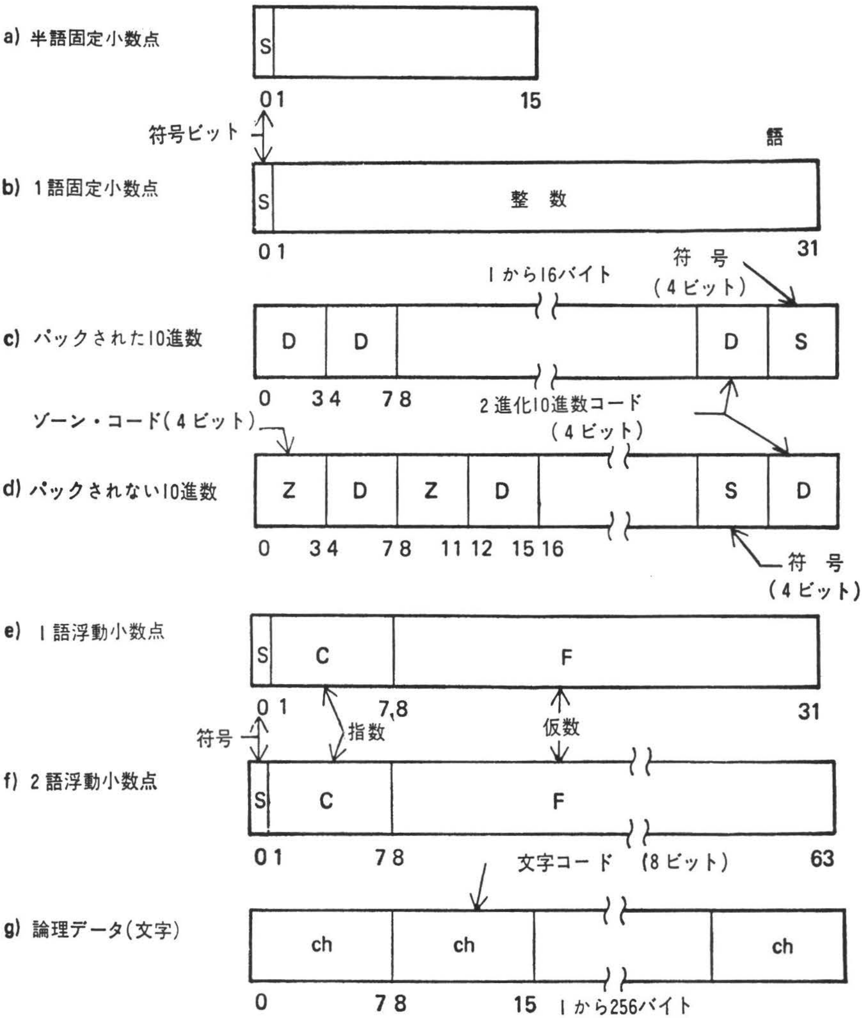


図 2.3 IBM 360 および 370 のデータ書式

できる．すなわち，記憶装置の中に格納されているビットの集合は IBM 360 プロセッサによって幾とおりかに解釈される． IBM 360 が2バイトの内容を整数（**図 2.3a**）として解釈すると，第1ビットは符号として，残り15ビットを2進数として解釈する（例えば，0000 0010 0001 1101 は10進数の＋541 に等

16進	2進	10進
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

図 2.4　16進—2進—10進数変換

しい2進数として解釈される）．IBM 360 が2バイトをパックされた10進数（**図 2.3c**）と解釈すると，第1バイトは2つの BCD 符号の数字として，第2バイトの始めの4ビットもBCD符号の数字として，そして残りの4ビットを符号と解釈する．（例えば，$\underbrace{0000}_{0}\ \underbrace{0010}_{2}\ \underbrace{0001}_{1}\ \underbrace{1101}_{\text{符号}}$ は10進数−021と解釈される）．

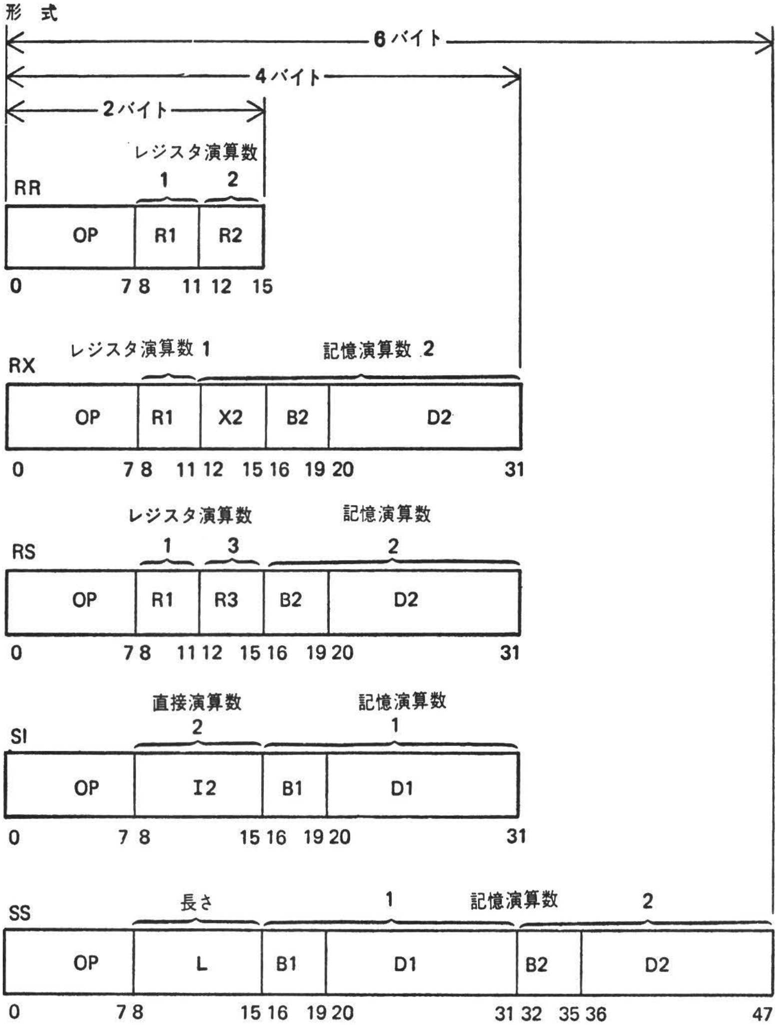
すべてのデータと命令は，物理的には一連の2進法の1と0とで格納されている．（したがって，10進の値+300を持つ16ビットの固定長半語は2進法で‘0000 0001 0010 1100’として格納される．便宜上，2進数は通常16進数（基数16）にて書かれる．**図 2.4** に **16進数** (hexadecimal) を示しておく．1桁の16進数の数字は4桁の2進数で置換えることができ，逆も可であることに注意してほしい．したがって，10進数の+300は2進数および16進数では次のように表わされる：

$$\left.\begin{array}{l} \text{B}'\underbrace{0000}_{}\ \ \underbrace{0001}_{}\ \ \underbrace{0010}_{}\ \ \underbrace{1100}_{} \\ \text{X}'\ \ \ 0\ \ \ \ \ \ \ \ 1\ \ \ \ \ \ \ \ 2\ \ \ \ \ \ \ \ \text{C} \end{array}\right\}\ \begin{array}{l} \text{2進数} \\ \text{16進数} \end{array}$$

接頭語のXとBは表現の形式（16進と2進）を示している．

固定小数点の数は半語または1語に格納される（**図 2.3a** および **図 2.3b**）．

IBM 360 では10進形式で数を格納することができる（**図 2.3c** および **図 2.3d**）．すなわち，数は2進数としてでなく，10進数表現にきわめて類似した形で格納される．例えば，数12は，最初の4ビットで数字1（0001）を，次の4



略記号：

OP ～ 命令コード
Ri ～ 演算数として使用される汎用レジスタ
Xi ～ インデックスとして使用される汎用レジスタ
Bi ～ ベースとして使用される汎用レジスタ
Di ～ 変　位
Ii ～ 直接演算数
L ～ 演算数の長さ

図 2.5 IBM 360 の命令の基本形式

ビットで数字2（0010）を表わして，1バイトに格納される．10進形式は事務用データ処理には便利である．

IBM 360 で，浮動小数点の数，論理データおよび文字列（**図 2.3g**）は記憶装置の中で**図 2.3** に示したような形に表現されている．

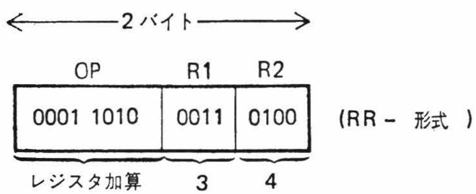
4. 命　　令

IBM 360 は算術，論理，制御移行および特殊割込みの命令を持つ．

IBM 360 の命令の形式は，**図 2.5** に示されている．これら5つの形式の命令は，基本的には使用する演算数の型によって異なっている．

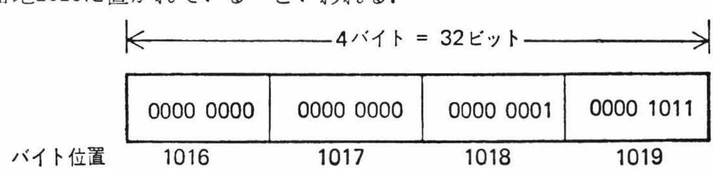
レジスタ演算数は16個の汎用レジスタ（32ビットの長さ）の1つに格納されているデータを参照し，命令の4ビットのフィールドによって番地指定される．レジスタは通常高速の電子回路で構成されており，コア記憶装置よりもデータに対するアクセスが速い．

例えば，**レジスタ加算命令は**次の形式である：



この命令は汎用レジスタ4（32ビット）の内容を汎用レジスタ3（32ビット）の内容に加えその和を汎用レジスタ3に残す．

記憶演算数はコア記憶装置に格納されたデータを参照する．演算数の長さは（**図 2.3** の説明のような）特定のデータの型により定まる．1バイトより長い演算数は最も値の小さい番地のバイト（論理的には左端）の番地で指定される．例えば，次に示したように場所1016，1017，1018および1019に格納された値，+267を持つ1語32ビットの2進固定小数点数（16進数で X '00 00 01 0B'）は"番地1016に置かれている"といわれる．



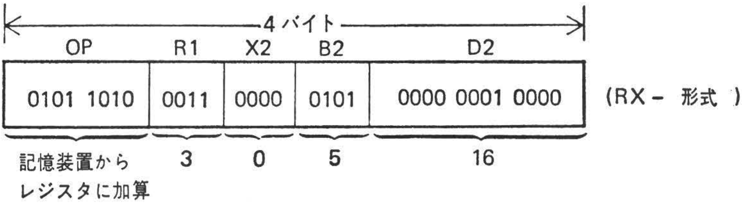
記憶演算数の第 i 番目の番地は命令から次のようにして計算される：

$$\text{番地} = c(Bi)+c(Xi)+Di \qquad (\text{RX形式})$$

$$\text{または}\ c(Bi)+Di \qquad (\text{RS, SI, SS 形式})$$

ここで c(Bi) および c(Xi) は，それぞれ汎用レジスタ Bi および Xi の内容を示す．例外として Xi＝0 のときは c(Xi) は 0 であるとみなされる．Bi＝0 のときも同様である．

例えば，汎用レジスタ 5 が数1000を持っているとき次の命令

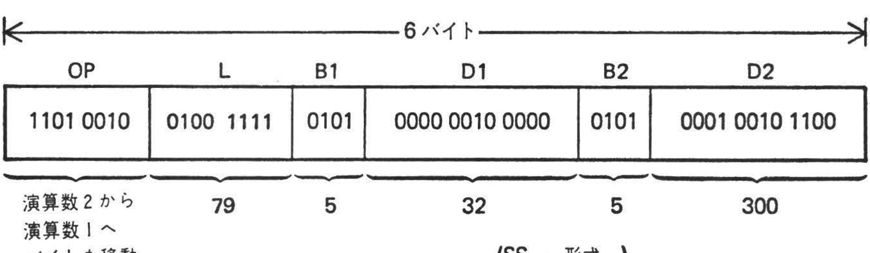


は

$$\begin{aligned}\text{番地} &= c(B2)+c(X2)+D2\\ &= c(5) \ \ +c(0) \ \ +16\\ &= 1000 \ \ +0 \ \ \ \ +16\\ &= 1016\end{aligned}$$

に置かれている 1 語（32ビット）の内容を汎用レジスタ 3（32ビット）に加え，その結果を汎用レジスタ 3 に残す．

再び汎用レジスタ 5 が1000を格納しているとし，別の例として次の命令を示す．（ＳＳ命令で長さは転送されるデータの長さより 1 だけ小さい値である．



例えば，長さ＝０は１バイトを転送する.)

この命令は２つの記憶演算数を持つ：

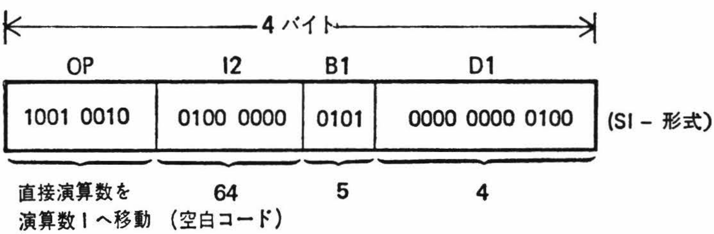
記憶演算数１の番地 ＝c(B1)＋D1＝c(5)＋32
＝1000 ＋32 ＝1032

記憶演算数２の番地 ＝c(B2)＋D2＝c(5)＋300
＝1000 ＋300＝1300

この命令は記憶場所1032〜1111から1300〜1379へ80バイトを複写（移動）する．１文字は１バイトで格納されるから（**図 2. 3g** 参照)，この命令は80文字の“カード・イメージ”を１つの領域から他へ複写するものと考えることができる．

直接演算数は命令の１部として格納されている１バイトのデータである．

再びレジスタ５が1000を格納しているとし，次の SS 命令を考えると，バイト 0100 0000（命令のビット８から15まで）が記憶場所1004に格納される．



代表的な IBM 360/370 の命令

本書の中では各種の IBM 360 命令が必要に応じて例や問題の中で使用されている．次に示す命令は特にその目的に関係のあるもので，適当な参照マニュアルで学習しておいてほしい．

5. 特殊機能

IBM 360 は 2,048 バイトのブロックでハードウェアによる記憶保護を行なっている．また，第９章で論じる精巧な割込み機構を有している．

	16進命令コード	記 号	意味(形式)
		ロード	
ロードー記憶装置ーレジスタ	58	L	ロード(RX)
	48	LH	半語ロード(RX)
	98	LM	可変長ロード(RS)
	18	LR	ロード(RR)
	12	LTR	ロード判定(RR)
		格 納	
	50	ST	格 納(RX)
	40	STH	半語格納(RX)
	90	STM	可変長格納(RS)
		加 算	
固定小数点演算	5A	A	加 算(RX)
	4A	AH	半語加算(RX)
	1A	AR	加 算(RR)
		比 較	
	59	C	比 較(RX)
	49	CH	半語比較(RX)
	19	CR	比 較(RR)
		除 算	
	5D	D	除 算(RX)
	1D	DR	除 算(RR)
		乗 算	
固定小数点演算	5C	M	乗 算 (RX)
	4C	MH	半語乗算 (RX)
	1C	MR	乗 算 (RR)
		減 算	
	5B	S	減 算 (RX)
	4B	SH	半語減算 (RX)
	1B	SR	減 算 (RR)

論理演算		比　較	
	55	CL	論理比較（RX）
	D5	CLC	論理比較（SS）
	95	CLI	論理比較（SI）
	15	CLR	論理比較（RR）
		移　動	
	D2	MVC	移動（SS）
	92	MVI	移動（SI）
		And	
	54	N	AND (RX)
	D4	NC	AND (SS)
	94	NI	AND (SI)
	14	NR	AND (RR)
		Or	
	56	O	OR (RX)
	D6	OC	OR (SS)
	96	OI	OR (SI)
	16	OR	OR (RR)
		排他 or	
	57	X	排他-or（RX）
	D7	XC	排他-or（SS）
	97	XI	排他-or（SI）
	17	XR	排他-or（RR）
		桁送り	
	8D	SLDL	左桁送り（2語論理的）（RS）
	89	SLL	左桁送り（1語論理的）（RS）
	8C	SRDL	右桁送り（2語論理的）（RS）
	88	SRL	右桁送り（1語論理的）（RS）
制御移行		結　合	
	45	BAL	結合分岐（RX）
	05	BALR	結合分岐（RR）
		分　岐	
	47	BC	条件分岐（RX）
	07	BCR	条件分岐（RR）
	46	BCT	計数分岐（RX）
	06	BCTR	計数分岐（RR）

雑

9E	HIO	I/O 停止 (RX)
41	LA	番地ロード (RX)
9C	SIO	I/O 開始 (RX)
0A	SVC	スーパバイザ呼出し (SI)
9D	TIO	I/O 検査 (RX)
43	IC	文字挿入 (RX)
91	TM	マスク判定 (SI)
42	STC	文字格納 (RX)

2.2 機 械 語

本節では機械語（計算機で実行される実際のコード）について論じる．再び，例を IBM 360 型計算機から選ぶ．しかしながら，これらは容易に他の計算機に適用できる．

本節では，読者が独自に機械語を学び始めるきっかけを作る．本節を読み終えた後に，読者は，自分の使用する計算機の機械語を述べた本やマニュアルを参照してほしい．

ここでは機械語を1と0とで書いたり16進数で書いたりせず，覚えやすい記号名による表現形式を用いる．

図 2.6 は CPU でロード命令として解釈される1と0との並びと，この命令を表わすのに使用する記号名による表現形式を示している．

次の簡単な例は機械語の特徴を説明するために本章で何回か使用される．

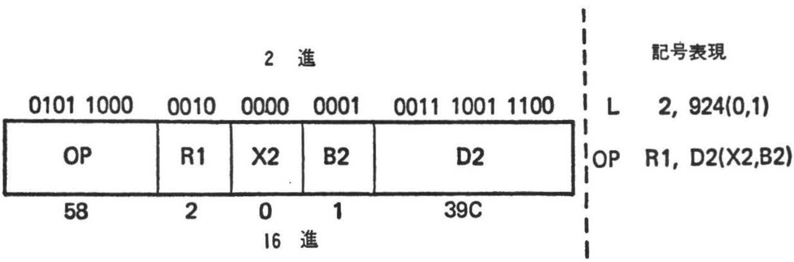


図 2.6 機械語の記号表現

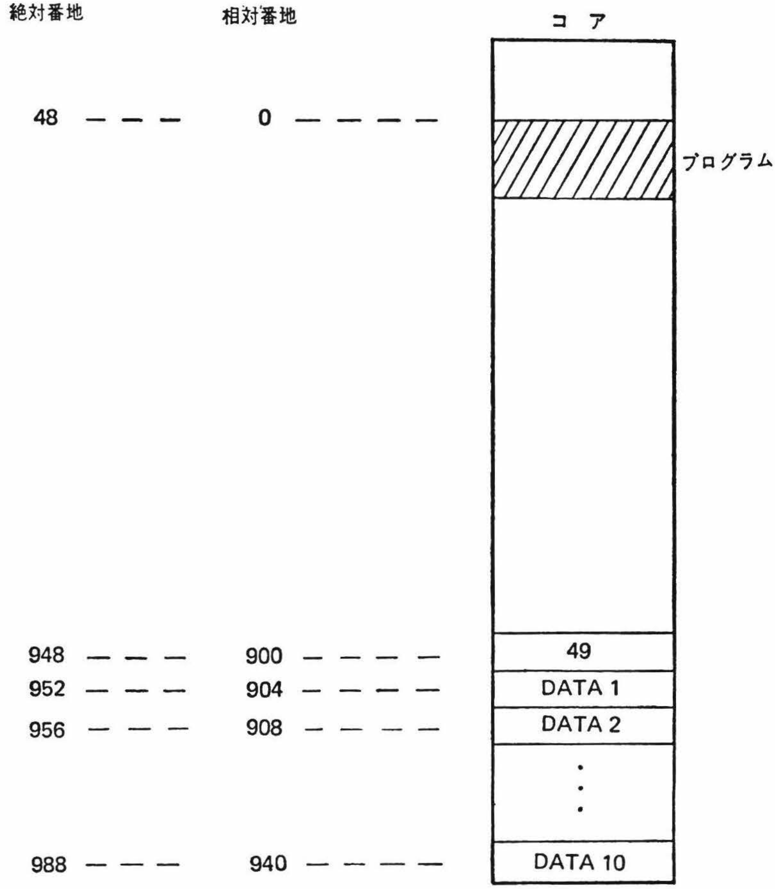


図 2.7　加算プログラムのコア割当て

次の仮定のもとに，数49を記憶装置の10個の連続した1語長語の内容に加えるプログラムを書け：

仮定1.　加算される10個の数はコアの絶対番地 952 から始まる連続した1語長の語に入っている.

仮定2.　プログラムはコアの絶対番地48から始まる場所に入っている.

仮定3.　数49は絶対番地 948 の1語長の語に入っている.

仮定4.　レジスタ1は値48を格納している.

絶対番地	相対番地	16進	命令	註釈
48	0	58201388	L 2,904(0,1)	番地 904+C(レジスタ1)=952 からレジスタ2にロードする
52	4	5A201384	A 2,900(0,1)	49を加算(番地=900+C(レジスタ1)=948)
56	8	50201388	ST 2,904(0,1)	格納
60	12	5820138C	L 2,908(0,1)	次のデータをロード
64	16	5A201384	A 2,900(0,1)	49を加算
68	20	5020138C	ST 2,908(0,1)	格納
⋮	⋮	⋮	⋮	
948	900	00000031	49	
952	904	⋮	⋮	

図 2.8 加算問題のプログラム——直接的な方法

コアは図2.7に示したような割当てが行なわれていると考えることができる.

2.2.1 長い方法—ループ無し

図2.8はこの加算を行なうプログラムを示す.

最初の命令L 2,904(0, 1) は1番目の数をレジスタ2にロードする. レジスタ2はアキュムレータとして使用される. 第2.1.3節で説明したように, IBM 360の番地は(オフセット+インデックス・レジスタの内容+ベース・レジスタの内容)により決定される. この命令においてはインデックス・レジスタは0である. レジスタ0は存在するが, これがインデックスやベースまたは分岐レジスタとして使用されるときは内容0を持つと仮定される. したがって, 上述の最初のロード命令で指定される番地は904+レジスタ1の内容(48), すなわち, 952である. この値は第1のデータ要素DATA1の絶対番地である.

プログラムの次の命令は絶対番地948の内容をレジスタ2に加える. 絶対番地948は49を格納している. 次にレジスタ2の内容を絶対番号952に書込む格納命令がくる. この命令は初めに入っていたデータを壊して, DATA1+49に等しい新しいデータで置換える. 同様にして次の3つの命令は49を DATA2に加える. 全く同じ3つの命令の組が各データ項目に対して使用される.

上のプログラムは正しく働くが, 幾つかの問題を持っている. 例えば, ただの10個ではなくて300個のデータ項目を処理しようとするとき, 命令に必要な記憶容量は(3命令)×(各命令の長さ)×(データ項目の個数)=3,600バイトとなる. したがって, 命令はコア中でデータの部分に重なってしまう. さ

らに，データ自身は 4×300＝1200 バイトの領域を占めるから，最初の命令から最後のデータの項目までの距離は 4,800 バイトになる．レジスタ1をベース・レジスタとして使用しても，第1のデータ項目と最後のデータ項目とに同時にアクセスすることはできない．プログラマは機械語でプログラムを書くとき，最大オフセットの値は $2^{12}-1$，すなわち，4,095（この値ではすべてのデータに到達できない）であることを留意しなければならない．（もちろん，2つ以上のベース・レジスタを使用することはできる．）

なお，前述のプログラムが48番地の代りに 336 番地にロードされても，レジスタ1の内容が 336 であれば正しく働く．このようにプログラムを別の場所に移すことを**再配置**という．ベース・レジスタを使用するとこの処理が容易になる．

2.2.2 命令をデータとして扱う番地修飾法

この例は**図 2.9** に示された“プログラム”に類似している．MIT の学生が女の子とデートするとしよう．この学生は**図 2.9** に示したことを行なうプログラムを書く．

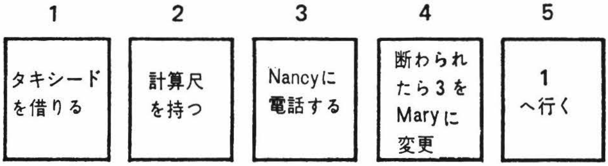


図 2.9 状　況

図の箱は記憶場所を示し，箱の中の言葉はプロセッサ，この場合は MIT の学生に対する命令を示す．プログラムは学生にタキシードを借りさせ，計算尺を持たせ（これは MIT を意味する），Nancy に電話させる．もし Nancy が断れば，MIT の学生は新しいプログラムを書くことを望まず，単に場所3の内容を Nancy から Mary に変更して1から4を繰り返す．場所4の命令を実行すると場所3の命令が変更される．もちろん，もし Mary が断れば，このプログラムでは学生はたくさんのタキシードを借りることとなり，Mary にはうるさいほど電話がかかってくることになる．

しかし，この MIT の学生は基本的な2つのプログラム技法を体得している．第1は，場所3の命令はデータとして処理できるということである．第2は，ステップ5の移行の規則によってループを作ることである．本節と次節ではこの2つの技法によって前のプログラムがどのようにして簡単化されるかを示そう．

図2.8のプログラムでは単に3つの命令を繰り返し使用していたにすぎない．変化したのはロードおよび格納命令のオフセットだけであった．最初の3つの命令の組でオフセットは904であり，次の組では908，次は912であった．代りの技法はこれら3つの命令と，ロードおよび格納命令のオフセットに4を加えて変更する一連の命令を続けただけのことである．計算機は3つの命令を実行し，2つの命令を変更してオフセットに4を加え，元に戻って3つの命令の組を再実行する．もちろん，2回目以降は違ったオフセットを持ち，したがって違った番地を参照する．

図2.10のプログラムは，命令自体を修飾してコアの10個所に数49を加える演算を実行する一連の命令を示す．

絶対番地	相対番地	命 令	註 釈
48	0	L 2,904(0,1)	データに49を加算する
52	4	A 2,900(0,1)	
56	8	ST 2,904(0,1)	
60	12	L 2,0(0,1)	ロード命令の変位を4増加する
64	16	A 2,896(0,1)	
68	20	ST 2,0(0,1)	
72	24	L 2,8(0,1)	格納命令の変位を4増加する
76	28	A 2,896(0,1)	
80	32	ST 2,8(0,1)	
		相対番地0に9回分岐する	
⋮	⋮		
944	896	4	
948	900	49	
952	904	データ	
⋮	⋮		

図 2.10 命令を修飾する加算問題のプログラム

付加的な仮定：

仮定5．相対番地896は値4を格納している．

プログラムがどのように働くかを見るためには，48番地の内容は L 2,901 (0,1) でなく**図2.11**のようになっていることに留意しなければならない．

命令のオフセットは記憶場所に格納された数の右端に置かれた部分である．この命令は1個のデータとみなすことができ，4を加えるとオフセットが更新できる．

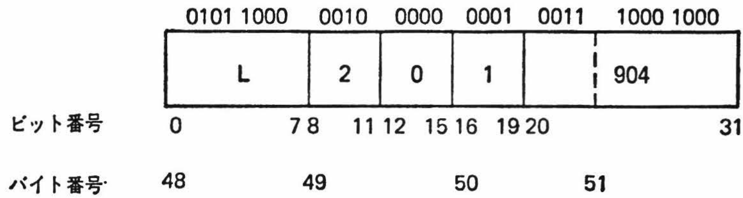


図 2.11　48番地の内容

命令をデータとして取り扱うことは，長期にわたってプログラムを保存しようとするとき元のプログラマが何をしようとしたのかを理解することが困難となるので，良いプログラム技法でない．多重プログラミング・システムの場合には，手続き（procedure）は自分自身を変更しないという純手続き（pure procedure，**再入可能，リエントラント・コード**：re-entrant code）の規則に違反する．ここでは命令の形成を説明し，命令もデータの一種として格納されるということを示す目的でこの例を用いた．

2.2.3 インデックス・レジスタを用いた番地修飾法

この例題を解く最も優れた方法は番地修飾のためにインデックス・レジスタを用いることである．番地は（オフセット＋ベース・レジスタの内容＋インデックス・レジスタの内容）であることを思い出してほしい．ここでも同じ3つのロード命令，49を加える命令，格納命令を使用する．これら3つの命令をループで繰返し，各パス毎に1つのインデッス・レジスタの内容に4ずつ加え，ロードおよび格納命令で指定される番地を更新する．次のプログラムがこの技法を用いている：

絶対番地	相対番地	命令	註 釈
48	0	SR 4,4	レジスタ 4 をクリアする
50	2	L 2,904(4,1)	配列のデータをロードする
54	6	A 2,900(0,1)	49を加える
58	10	ST 2,904(4,1)	データを格納する
62	14	A 4,896(0,1)	4 をインデックス・レジスタに加える
		9 回相対番地 2 に分岐する	

このプログラムの最初の命令は減算命令である．これはレジスターレジスタ命令で，レジスタ 4 からレジスタ 4 を引くのでその内容を 0 に初期化する．なお，この命令の長さは 2 バイトである．前節でも論じたように IBM 360/370 は長さの異なる命令を持っている．SR 命令は絶対番地48から始まっている．次の命令は50番地から始まっている．また，ロードおよび格納命令はインデックス・レジスタとしてレジスタ 4 を指定している．最初ループを通るときレジスタ 4 は値 0 を持ち，次回は 4，……．こうしてロードおよび格納命令は毎回違ったデータを指定する．

さて，どのようにして命令の番地修飾ができるかわかったので，次に実際の

絶対番地	相対番地	命 令		
48	0	SR	4,4	
50	2	L	2,904(4,1)	49 を加算
54	6	A	2,900(0,1)	
58	10	ST	2,904(4,1)	
62	14	A	4,896(0,1)	インデックス・レジスタに 4 を加算
66	18	L	3,892(0,1)	レジスタ 3 にtempをロード
70	22	S	3,888(0,1)	1 を減算
74	26	ST	3,892(0,1)	tempへ格納
78	30	BC	2,2(0,1)	結果が正のとき分岐 (2 は条件コードを示す)
—	—	—		
—	—	—		
—	—	—		
936	888	1		
940	892	(初期値 10，各ループで 1 ずつ減ぜられる)		
944	896	4		
948	900	49		
952	904	データ		

図 2.12 ループを持つ加算問題のプログラム

ループを作るための命令を追加することにする.

2.2.4 ループの構成

本節では機械語を用いて2つのループ構成の手法を論じる. ここでさらに2つの仮定を追加する：

仮定6.　相対番地 892 は値10を持つ.

仮定7.　相対番地 888 は値 1 を持つ（第1法のみ).

図 2.12 は1つのループ技法を示す.

最初の 4 つの基本命令が実行された後に，仮りの場所から1を減じその結果が正か否かを判定する一連の命令が置かれている. 正であれば制御はプログラムの相対番地2に戻る.

1つの命令で**図 2.12** の最後の 4 つの 命令の働きを行なう 計数分岐命令（**図 2.13** に示された BCT）がある.

BCT 命令の説明について読者はマニュアルを参照してほしい. レジスタ 3

絶対番地	相対番地	命令		
48	0	L	3,892(0,1)	レジスタ 3 に10をロード
52	4	SR	4,4	レジスタ 4 を消去
54	6	L	2,904(4,1)	
58	10	A	2,900(0,1)	49を加算
62	14	ST	2,904(4,1)	
66	18	A	4,896(0,1)	インデックス・レジスタに 4 を加算
70	22	BCT	3,6(0,1)	レジスタ 3 から 1 を減じ，正ならば相対番地 6 へ分岐
⋮	⋮	⋮		
940	892	10		
944	896	4		
948	900	49		
952	904	データ		
⋮	⋮			
992	未使用			

図 2.13　例題の最終版

は0になるまで1ずつ減ぜられる．レジスタの内容が正の間，番地部で指された番地に制御が移される．この場合には6＋（レジスタ1の内容）に移る．0に達すると分岐は起こらない．たいていの計算機は同じような分岐命令を持っている．

最初のプログラムで命令に120バイト，データに44バイト使用したのに対して，プログラムは命令が26バイト，データが52バイトとなり，86バイトの節約となる．なお，前述のプログラムはすべて，48番地でなく例えば400番地など，コアの任意の場所に置くことができる．この場合レジスタ1のみ変更すればよい．

2.3 アセンブラ語

利用者が計算機と通信しようとするとき各種の言語が使用できる：

英語	プログラマには最適
PL/I，FORTRAN	↑
・	│
・	│
・	│
アセンブラ語	│
記号名による機械語	↓
機械語	計算機に最適

これまでは最も低いレベルの言語を用いた．ここでは今日最も計算機に依存する言語としてプログラマに使用されているアセンブラ語に進む．

アセンブラ語を用いることの利点は主として4つある：

1. 覚えやすい記号名である．例えば格納命令にビットの列01010000と書く代りにSTと書く．
2. 番地は記号名であり，絶対値でない．
3. 解読がより容易．
4. プログラムに対するデータの導入がより容易．

アセンブラ語を用いることの欠点は原始プログラムから目的コードに翻訳するためにアセンブラを使用しなければならないことである．IBM 360/370 アセンブラ語の多くの特徴は他の計算機に対するアセンブラ語にも存在する．例は読者の使用している計算機にも容易に変換できる．

2.3.1 アセンブラ語のプログラム

前節で論じたプログラム（図2.14）をアセンブラ語で書いてみよう．このときには，機械語でプログラムを書いたとき行なわれた仮定は不要である．その1つの仮定は，プログラムの開始番地はコアの絶対番地48であるというものであった．プログラマとしてはコアの中でプログラムがロードされる場所を仮定することはできない．したがって，実行に先立ってコア中のプログラムの番地をベース・レジスタにロードする方法が必要である．すなわち，実行時になってから始めてプログラム，プログラマまたは他の人は，ローダが利用者のプログラムをコアのどこにロードするかを知ることができる． BALR 命令はベース・レジスタをロードするための1つの機構である．

プログラム			註　釈
TEXT	START		プログラムに名前を付ける
BEGIN	BALR	15,0	レジスタ15を次の命令の番地に設定する
	USING	BEGIN+2,15	レジスタ15がベース・レジスタであり，その内容が次の命令を指していることをアセンブラに知らせる擬似命令
	SR	4,4	レジスタ4を帰零（index=0）
	L	3,TEN	数10をレジスタ3にロード
LOOP	L	2,DATA(4)	DATA(index)をレジスタ2にロード
	A	2,FORTY9	49を加算
	ST	2,DATA(4)	DATA(index)に格納
	A	4,FOUR	レジスタ4に4を加算(index=index+4)
	BCT	3,LOOP	レジスタ3から1を減じ，0でなければLOOPに分岐
	BR	14	呼出者に復帰
TEN	DC	F'10'	数10
FOUR	DC	F'4'	数4
FORTY9	DC	F'49'	数49
DATA	DC	F'1,3,3,3,3,4,5,8,9,0'	処理すべきデータ
	END		

図 2.14　アセンブラ語のプログラム

アセンブラが自動的に命令の変位部を計算するためには，どのレジスタがベース・レジスタとして使用可能であり，そのレジスタの内容が何かを知る必要がある．USING 命令はアセンブラにこれらのことを告げ，正しいコードを作り出すことを可能とする．USING 命令は**擬似命令**（pseudo-op）である．擬似命令はアセンブラの処理を規定するアセンブラ語の命令であり，機械の命令をアセンブラに示す**機械命令**（machine-op）とは区別される．定数定義（Define Constant：DC）および領域定義（Define Storage：DS）命令は，アセンブラに値 10, 4 および 49 を記憶装置の中の 3 つの連続した語（"F"）に置き，データのためにもう10語をとっておくことを指示する擬似命令である．Fの前に置かれた数は多重割付けを指定する．例えば，100 F のときアセンブラは100語の領域を確保する．

説　明

1. **USING** は，アセンブラにどの汎用レジスタをベースとして用い，その内容は何かを示す擬似命令である．IBM 360 計算機では番地指定用として特別のレジスタを割当てていないので，プログラマがアセンブラに，どのレジスタをどのように用いるかを通知するためにこの擬似的命令が必要である．番地は相対値である，この擬似命令によってプログラマはベース・レジスタに設定してある番地をアセンブラに通知することができる．したがって，アセンブラは正しいベース・レジスタとオフセットを用いて機械コードを作ることができる．
2. **BALR** は次の番地をレジスタにロードし，第2フィールドの番地に分岐するための計算機の命令である．この例のように，第2演算数がレジスタ 0 ならば，引続いて次の命令が実行される．ベース・レジスタをロードする BALR とベース・レジスタに何が入っているかをアセンブラに通知する USING との相違を完全に理解しておいてほしい．この区別はアセンブラを学んだ後ではより明確となろう．しかしここでは，USING はアセンブラに情報を与えるが，レジスタをロードしないことに注意してほしい．したがって，USING が指定している番地をレジスタが持っていないとしたら，プログラムに誤りが発生する．
3. **START** は，プログラムの始まりがどこかをアセンブラに告げ，利用者

がプログラムに名前を付けることができるようにする擬似命令である．この例では名前は TEST である．

4. **END** は，プログラムの最後のカードに達したことをアセンブラに告げる擬似命令である．

5. **図 2.8**の例のように命令の演算数部に番地の値を入れる代りに，この例では記号名が入っていることに注意．アセンブラが作られた主たる理由はプログラマが計算機の番地を計算するという負担を除くことにあった．

6. 最後の機械命令 BR 14は汎用レジスタ14に入っている番地へ分岐する．規定によって呼出しプログラムは復帰番地をレジスタ14に残しておく．

2.3.2 リテラルの使用例

ここでは同じ例を**リテラル**を使用して繰り返す．リテラルとは，プログラマが要求する定数を含んだデータ領域をアセンブラによって作るための機構である．

図 2.15 のプログラムにおいて，=F′10′，=F′49′，=F′4′ は値 10, 49, 4 を含むデータ領域を作るリテラルである．これらはデータの番地の代りに用いられている．

アセンブラは命令L　3, =F′10′ を翻訳して番地部は値10を持つ語を指すよう

```
TEST      START   0
BEGIN     BALR    BASE,0
          USING   BEGIN+2,BASE
          SR      4,4
          L       3, = F'10'
LOOP      L       2,DATA (4)
          A       2, = F'49'
          ST      2,DATA (4)
          A       4, = F'4'
          BCT     3, *-16
          BR      14
          LTORG
DATA      DC      F'1,3,3,3,3,4,5,8,9,0'
BASE      EQU     15
          END
```

図 2.15　リテラルを使用したアセンブラ語のプログラム

にする．普通にはアセンブラは“リテラル表”をプログラムの終りに作成する．この表はリテラルを用いて使用されたすべての定数を含んでいる．しかしながら，擬似命令 LTORG は今までに出てきたりリテラルをもっと前に置くようにアセンブラに指示する．この擬似命令は，非常に長いプログラムの場合などで用いられる．例えば，DC 命令が10,000個のデータを作るとすると，ロード命令のオフセットは置かれたリテラルに届くことができない．この場合，DC 命令の前にリテラルを置くようにしたい．

同じプログラムの中の BCT 命令で番地＊－16を使用した．＊記号は“その場所”を意味する記号である．表現＊－16は‘現在の命令の番地引く16’の番地，すなわち LOOP を参照する．（この種の番地指定法は通常好ましくない．LOOP と BCT の間に他の命令文を挿入するとき，プログラマは忘れずに16を変更しなければならない.）

アセンブラ語プログラム			相対番地	アセンブルされた記号表現のプログラム	
TEST	START				
BEGIN	BALR	15,0	0	BALR	15,0
	USING	BEGIN+2,15			
	SR	4,4	2	SR	4,4
	L	3, =F'10'	4	L	3,30(0,15)
LOOP	L	2, DATA (4)	8	L	2,42(4,15)
	A	2, =F'49'	12	A	2,34(0,15)
	ST	2, DATA (4)	16	ST	2,42(4,15)
	A	4, =F'4'	20	A	4,38(0,15)
	BCT	3, *-16	24	BCT	3,6(0,15)
	BR	14	28	BCR	15,14
	LTORG				
			32	10	
			36	49	
			40	4	
DATA	DC	F'1,3,3,3,3,4,5,9,0'	44	1	
			48	3	
			52	3	
				⋮	
	END				

図 2.16 例題プログラムのアセンブラ語版

文 BASE EQU 15は記号 BASE に値15を割当てる．BASE は15であるとみなされる．EQU 擬似命令を用いてプログラマは変数を定義することができる．例えば，別のベース・レジスタを使用したいときには EQU 文のみを変更すればよい． EQU 文の演算数として，算術式が許される．

図 2.16 には前のプログラムのアセンブルされた形（すなわち，アセンブラの出力）が示されている．

ある種の擬似命令（例えば START や USING）は機械語を生成しない．また命令 BR 14は BCR 15, 14 に翻訳されている．この理由は BR 命令が，アセンブラの命令記号の集合の１つであり，これによってプログラマは特定のマスクが後に続く BC 命令の代りに，もっと多くの種類の記号を使用できるようにしているからである.

2.4 まとめ

新しい計算機を理解しようとするとき用いられる一般的な方法を示し，これを IBM 360 および 370 に応用した．機械語プログラムの例を発展させてベース・レジスタの使用，命令とデータの記憶，インデックス修飾とループ構成を説明した． IBM 360 基本アセンブラ語 (Basic Assembly Language : BAL) の幾つかの，例えば，記号番地法，リテラルおよび擬似命令などの特徴を紹介した．

演習問題[4]

1. すべてのデータは，一定の法則にしたがって解釈すると意味を持つような１と０によって記憶されている．指定された条件にしたがって次の 0, 1 の列の組を解釈せよ：

a.	OP1	1001	0111	1001	1100
	OP2	0110	1001	0010	1100

4　＊は IBM 360マニュアルの使用を要する問題であることを示す．
†は計算機固有の問題であることを示す．

OP3　1000　0011　0110　1101

1) 半語の整数

2) パックされた10進数

b. OP1　0100　1110　1111　0110

OP2　0110　0000　1111　0001

OP3　1111　0101　1111　0010

1) 半語の整数

2) 文字 (EBCDIC)

2. 次に示すのはコアの語および半語を示す幾つかの16進表現である：

(1) 052C　(2) 452C

(3) 4528367D　(4) 5914973C

各々に対して

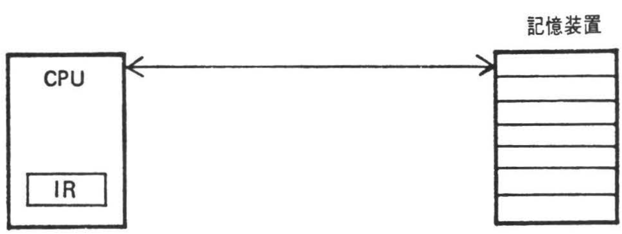
a. 等価な2進数を記せ.

b. 各々が半語／1語の整数のとき，10進数の値を記せ.

c. 各々がパックされた10進数のとき，10進数の値を記せ.

d. 正しい IBM 360 機械命令とするとき，記号による表現を示せ．できないときは理由を記せ.

3. この問では計算機構成の基本的な性質を考察する．例の簡単な計算機は，命令とデータを格納する番地の付けられた記憶装置と，命令を解釈し指定された動作を行なう中央演算処理装置 (CPU) とから成っている．CPU は現在解釈されている命令を保存する命令レジスタ (IR) を持っている.



この計算機の命令は次の形式である：

命令コード	演算数1	演算数2	結　果	次の命令

問a〜dに対して，命令の長さを短かくするという目的で，基本的な計算機を設計し直せ．これらの問に対して，記憶場所Aの数を記憶場所Bに加え結果をCに格納する次の命令を考えよ．次に実行される命令はNEXTにあるとする．

ADD	A	B	C	NEXT

a. NEXT 部を取り除く方法を示せ．元の計算機では不要であったが，新しい計算機で追加せねばならない命令は何か．CPU にはどのようなレジスタを追加しなければならないか．

b. RESULT 部を取り除く方法を示せ．(CPU にレジスタを追加してはならない.)元の計算機では不要であったが，新しい計算機で追加せねばならない命令は何か．

c. RESULT 部を取り除く方法は CPU に記憶装置の語と同じ大きさの1つのレジスタを追加することである．このようなレジスタは通常アキュムレータ（AC）と呼ばれる．この場合に追加される命令は，アキュムレータの内容を記憶装置の指定された語に記入する格納命令である．他のすべての命令は2つの演算数を使用し，結果をアキュムレータに置く．RESULT 部以外に1つの演算数部もまた取り除くことができる．どのようにしてこれを行なうかを述べよ．どんな命令を追加しなければならないか．

d. IBM 360や他の多くの計算機は，基本的な4番地形式の計算機ではなく，上の問で得た2番地形式の命令を使用している．命令の長さを短くするとどのような利点があるか．また欠点は何か．

4. **a.** レジスタの少ない，または用途の限定されている計算機に比して，IBM 360のようにレジスタの多いことの利点は何か．

***b.** IBM 360でレジスタ0は3つの場合に他のレジスタと異なった働きをする．各々の例を示し，それらについてレジスタ0がどのように使用されるかを説明せよ．

5. ***a.** BALR および USING 命令の機能的な相違点は何か．各々にアセンブル時に何が起こるか．実行時にはどうか．

b.　次の各プログラム・セグメントに対する等価な記号表現による機械語プログラムを書き，命令 LH　1,DATA 2 によってレジスタ 1 に置かれる値を求めよ．

```
OCT15   START   0           OCT15   START   0
        BALR    15,0                USING   *,15
        USING   *,15                BALR    15,0
        LR      10,15               LA      10,DATA2
        USING   *,10                USING   DATA2,10
        LH      1,DATA2             DROP    15
        BR      14                  LH      1,DATA2
DATA1   DC      H'1'        DATA1   DC      F'1'
DATA2   DC      H'2'        DATA2   DC      F'2'
DATA3   DC      H'3'        DATA3   DC      F'3'
        END                         END
```

6. *a.　次のプログラムは 2 を 3 倍する掛算を実行し，結果を1000番地に格納するものとして作られているが，正しく働くか．（注意：番地1000はコアの場所1000を示している．ベースまたはインデックス・レジスタは使用されていない．）

```
L       3,=F'2'
M       3,=F'3'
ST      3,1000
```

b.　次は10を 2 で割るか確かめよ．

```
L       3,=F'10'
D       2,=F'2'
ST      3,1000
```

7. *a.　次の相違点は何か．

```
INDEX   EQU     5
INDEX   DC      F'5'
```

b.　CR 命令と CLR 命令の違いは何か．

8. *a.　次の一連の命令を実行した後のレジスタの内容は何か．

```
LA      3,=A(XYZ)
LR      3,3
L       3,=F'5'
```

```
XYZ      LCR      3,3
         LNR      3,3
```

b.　BE SAME の次に実行される命令はどれか．その理由は何か．

```
         CLI      =F'3',3
         BE       SAME
         LR       3,5
SAME     AR       5,5
```

9.　*a.　どのようなとき ‘SRDA 0,5’ と ‘SRDL 0,5’ は違うように実行されるか．

b.　どのようなとき ‘SLL 1,1’ と ‘LA 1,0(1,1)’ は違うように実行されるか．

c.　どのようなとき，‘MVC TEMP(0),DATA’ と ‘MVC TEMP(1),DATA’ は違うように実行されるか．

d.　‘LA 1,0(1)’ は ‘何もしない’ ことと等価か．
‘SRA 1,0’ はどうか．違うなら説明せよ．

e.　STOMP が次のように定義されるとする．

```
STOMP    DC       C'ERASURE'
```

次の命令は各々どのように実行されるか．

```
         MVC      STOMP+1(8),STOMP
         MVC      STOMP(8),STOMP+1
```

10.　*IBM 360のすべての命令は CC（条件コード）を設定したり検査したりするか． IBM 360 命令の CC の役割りは何か．

11.　次の IBM 360の命令に対してマイクロ流れ図を描け．（図 2.2 参照）

a.　S（減算，RX 形式）

b.　BCR（条件分岐，RR 形式）

c.　BXLE（インデックス低等による分岐：Branch on Index Less than or Equal, RS 形式）

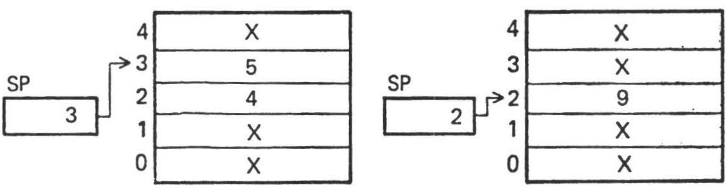
12.　“スタック”計算機と呼ばれる次の計算機システムの構成を考える：

2つの記憶装置を有する．1つの記憶装置は直接番地指定が可能である．

すなわち，任意の場所で随時読出したり書込んだりできる．他方の記憶装置は“スタック・ポインタ”と呼ばれるレジスタを持ち，“スタック・ポインタ“が現在指している場所でのみ読出したり書込んだりできる．“スタック・ポインタ“を SP と記す．

命令は3種類ある．(1) 2つの記憶装置の間でデータを転送する命令，(2)“スタック”記憶装置だけに作用する命令，および (3)制御移行命令．IBM 7044 や 360 のような AC（アキュムレータ）や GPR（汎用レジスタ）などのレジスタがないことに注意．

まず ADD 命令を論じよう．スタック記憶装置は次の(a)図に示された内容を持つと仮定する．



(a) (b)

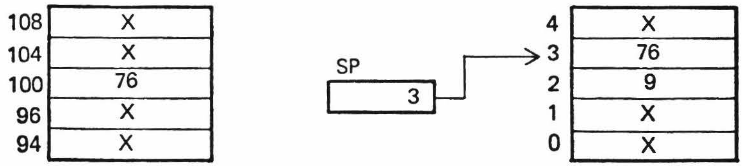
ADD 命令は“スタック”記憶装置を(b)図の内容に変える． Xと記された記憶場所は命令によって影響を受けず無視してよい．2つ動作が起こったことに注意せよ．(1) C(SP) および C(SP—1) で示された場所の内容が加算され，結果は C(SP—1) で示された場所に記入される．(2) SP は1だけ減ぜられる．

a. この計算機に対して構成図を描け（**図 2.1** 参照）．

b. ADD 命令のマイクロ流れ図を描け，この図は先の**図 2.2** の流れ図と同じ程度の複雑さとすること．実際のレジスタなどは必ずしも同じではない．使用する記号の定義を明確にしなければならない．命令は既に解釈レジスタに入っていると仮定しておく．（ADD 命令は演算数を持たず，命令コードのみを持っている．また ADD 命令は1バイトの命令コードだけが必要であると仮定する.)

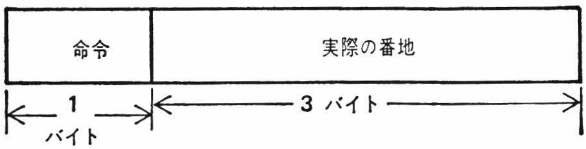
c. LOAD 命令は次に述べるように働く．通常の記憶装置は (c)図に示し

た内容を持っており，“スタック”記憶装置は(b)図のままであると仮定する．命令 LOAD 100 の実行結果は(d)図のようになる．



(c)　　(d)

この命令は C(SP+1) にて指定された場所にロードを行ない，C(SP) を1だけ増す．LOAD 命令は4バイト長とする．すなわち，



LOAD 命令に対してマイクロ流れ図を描け．

13. 次の IBM 360 をアッセンブラ語のプログラムは関数

$$A=2*B+2*C-1$$

の値を計算する．

```
 1)  COMPUTE   START
 2)            USING   *,15
 3)            L       1,B
 4)            SLA     1,1
 5)            L       2,C
 6)            SLA     2,1
 7)            AR      1,2
 8)            S       1,=F'1'
 9)            ST      1,A
10)            BR      14
11)  A         DC      F'0'
12)  B         DC      F'5'
13)  C         DC      F'7'
14)            END
```

a. 上のプログラムが正しく働くことを，命令を1つずつ机上で実行して，次の表に記入しながら確かめよ．各レジスタと記憶場所について各命令の実行後の内容を示せ．

命 令	レジスタ1	レジスタ2	場所A
3			
4			
5			
6			
7			
8			
9			

b. 上のプログラムはかなり効率の良いものである．記憶容量は44バイトで，遅い掛算命令とリテラル2を使用している．文3から文9の間だけを書き直すことによってプログラムを合計32バイトの長さのただの12文にすることができる．この新しいプログラムは同じ方法でなくてもよいが，上と等しい関数を計算しなければならない．

前のプログラムより効率の良い新しいプログラムを作るために，次のプログラムの空白の部分に記入せよ．最適なプログラムでなくてもよいし文3から文9のすべてを使用する必要もない．

```
 1)  COMPUTE   START
 2)            USING   *,15
 3)
 4)
 5)
 6)
 7)
 8)
 9)
10)            BR      14
11)  A         DC      F'0'
```

12)	B	DC	F'5'
13)	C	DC	F'7'
		END	

14. 任意のディジタル計算機のための メーカのマニュアルを入手して第2.1.1節の5つの基本的な問に答えよ．

15. †次の仕事を行なうサブルーチンを IBM 360 アセブラ語で書け．

a. 制御がそのプログラムに移されるとき，レジスタ2および3は8つのEBCDIC 文字（各レジスタに4つの8ビット文字）を持っている．

b. サブルーチンは EBCDIC の“ ,”の個数を数えてその数をレジスタ1に入れて返却する．

例：

	レジスタ2	レジスタ3
入力	A , B C	D E , F (EBCDIC)
	レジスタ1	
出力	00000002	(16進)

この問題の解は8～20枚のアセンブラ・カードを必要とする．

プログラムを評価するための採点プログラム（grader[5]）を用いるので，各プログラムの形式は次のとおりにすること．

GRADER	**STUDENT**	
レジスタ2,3をロードする	STUDENT START	0
L　15, =V(STUDENT)	ENTRY	STUDENTN
BALR 14,15	USING	*, 15
答えが正しいか検査する	プログラム本体	
	⋮	
	⋮	
	BR	14
	作業域	
	⋮	
	LTORG	

5　この問題は採点プログラムなしにでも解ける．この場合読者はデータと主プログラムを用意せねばならない．採点プログラムは学生のプログラムを呼出し，結果を記録する．採点プログラムは先生やこれを望む学生のために定義している．

```
STUDENTN EQU       *
         END
```

学生のプログラムはレジスタ14または15の内容を変更してはならない.

16. †すべてのデータは文字（EBCDIC）の形でアセンブラによって機械に読込まれる．したがって，アセンブラまたはコンパイラはプログラムが規定した内部表現にデータを変換するためのルーチンを用意しなければならない．

C，X，PおよびF形式に対して'DC'擬似命令を処理するとき，アセンブラにより行なわれるデータ処理を模擬する IBM 360 アセンブラ語のサブルーチンを書け．与えられるのは記憶装置の中に置かれている定まった長さの EBCDIC の文字列で，それを指定されたデータ形式に変換することが要求される．

a. **採点プログラム[5] との通信**　プログラムに入ったとき，汎用レジスタは次の情報を持っている．

レジスタ0：

レジスタ0の内容は実行すべき変換の型を示す：

0	文字
1	16進数
2	パックされた10進数
3	1語の2進数

レジスタ1：

レジスタ1の内容は変換されるデータの長さと記憶場所を示す．ビット0～7はデータのバイト数を，ビット8～31はデータの絶対番地を納めている．

個数	番地
0　　7	8　　31

レジスタ2～12：

レジスタ2～12の内容は重要ではない．しかしプログラムでは内容を保存し，復元しておかなければならない．

レジスタ13：

レジスタ13は汎用レジスタの内容を格納してもよい18語の領域の先頭番地を納めている．

レジスタ14：

レジスタ14は採点プログラムへの復帰先の番地を納めている．処理を終えたらこの番地に分岐しなければならない．

レジスタ15：

レジスタ15はプログラムへの入口の番地を納めている．プログラムが採点プログラムへ復帰するとき，レジスタ2～12の内容は呼出されたときと同じでなければならない．またレジスタ1は次の構造を持つプログラム中の語の番地を格納していなければならない．

個数	番地
0　　7	8　　31

“個数”は変換されたデータのバイト数，“番地”はプログラム中の変換されたデータを格納している領域の絶対番地である．

入力および出力データには次の制限が課せられている：

文字データの最大長は40バイトである．

16進数に変換されるデータの最大長は40バイトで偶数である．生じる答えの長さは最大20バイトである．（したがって，0で埋める必要はない．）

パックされた10進数に変換されるデータは，頭に付く正または負の符号を含まずに，最大16数字である．答えを返却するときはできる限り短かくしなければならない．すなわち，先行する0は取り除くこと．答えは EBCDIC の符号コードCおよびDのいずれかを付けて返却しなければならない．

また，1語の2進数に変換されるデータは正または負の符号を持つことがあり，最大長は16数字である．答えは1語境界に合せなければならない．

入力列をいささかたりとも変更してはならない．

b. **例** 入力として長さ3の文字列 ‘110’ が与えられたとすると，答えは次のようになる．

変換	内部コード（16進数）	長さ
C	F1F1F0	3
X	0110	2
P	11C0	2
F	0000006E	4

0の詰め合せと位置合せが X, P, F 変換で行なわれていることに注意．

17. †次の図のような基数の混合した加算および減算を行なう IBM 360 アセンブラ語のサブルーチンを書け．

レジスタ1および2は演算数を格納し，レジスタ0は答えを入れる．各レジスタの第1バイトは符号，第2バイトは日，第3は時間，第4は分である．

	1日	22時間	3分
−(		22時間	20分)
	0日	23時間	43分

レジスタ	内容（16進）			
R1	+	01	16	03
R2	−	00	16	14
R3	+	00	17	2B

学生のプログラムは問16と同様の形式を持たなければならない．

第 3 章

アセンブラ

アセンブラは，入力としてアセンブラ語のプログラムを受理し，等価な機械語のプログラムと，ローダに対する情報とを作成するプログラムである（**図 3.1**）．本章においてはアセンブラの設計を論じる．

ここでは機械語を作りだす手続きに焦点を合せるが，アセンブラは機械語のほかにローダの使用する他の情報をも作成しなければならないことを留意しておいてほしい．例えば，外部で定義された記号には註を付けてローダに渡さなければならない．というのはアセンブラはこれらの記号の番地の値を知らないので，これらの記号を含むプログラムを探し，コアにロードし，そして呼出しを行なっているプログラムの中でこれらの記号に値を設定することはローダに任されているからである．ローディングについては第 5 章で論じる．

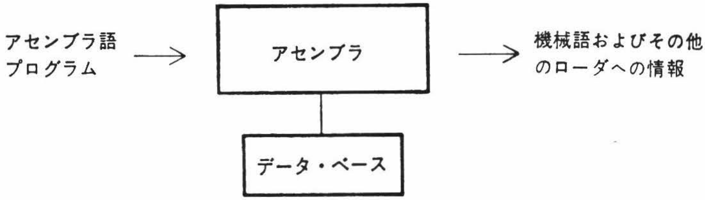


図 3.1　アセンブラの機能

本章においては主として機械語の生成について考える．説明のために用いる例は IBM 360 型のアセンブラと，記号にて表現した機械語を使用する．アセンブラの設計，および，その後のマクロ・プロセッサの設計においては多くのアルゴリズムが使用可能である．ここではこれらのプログラムの基本的な役割りをよく表現する方法を選ぶ．

本書では，“デック”と“プログラム”という言葉を使用している．かつて

“デック”とはカードのデックを意味した．今日では各種の2次記憶装置とタイプライタ端末の使用によって，プログラムは実際にカード・デックの形をとらないことが多い．“カード”は情報の手ごろな単位である．他の装置でも同様な，時として可変長の，単位である**レコード**への分割が可能である．これらの異った記憶形式は本質的に互換性がある．“文”は1枚のカードや他の形式のレコードとすることができるし，“カード”はテープやドラムの1つのレコードとすることもできる．本書や計算機業界で用いられているように，“デック”という言葉は計算機に各種の形式のプログラムを入力または出力するときよく用いられている．

3.1 一般的な設計手順

アセンブラ設計の詳細を論じる前に，ソフトウェア設計の一般的な問題点を調べてみよう．次にあげたのは設計者が従わなければならない6つのステップである：

1. 問題を設定する．
2. データ構造を規定する．
3. データ構造の書式を定義する．
4. アルゴリズムを規定する．
5. 機能単位（modularity）を探す（すなわち，1つのプログラムが独立なプログラム単位に分割できる可能性）．
6. モジュール毎に1から5を繰り返す．

本書ではアセンブラ，ローダおよびコンパイラの設計においてこの手順に従っている．

3.2 アセンブラの設計

3.2.1 問題の設定

われわれがアセンブラになって**図3.2**の左の列のプログラムを翻訳してみよ

う．START 命令を読込むと，これは，プログラムに JOHN という名前を付けている（アセンブラに対する）擬似命令であることがわかる．アセンブラはこの名前をローダに渡さなければならない．次の命令は USING 擬似命令である．これはアセンブラに，レジスタ15はベース・レジスタであり，実行時にプログラムの最初の番地を格納することを告げる．

原始プログラム			パス 1		パス 2	
			相対番地	記号表現の命令	相対番地	記号表現の命令
JOHN	START	0				
	USING	*,15				
	L	1,FIVE	0	L 1,−(0,15)	0	L 1,16(0,15)
	A	1,FOUR	4	A 1,−(0,15)	4	A 1,12(0,15)
	ST	1,TEMP	8	ST 1,−(0,15)	8	ST 1,20(0,15)
FOUR	DC	F'4'	12	4	12	4
FIVE	DC	F'5'	16	5	16	5
TEMP	DS	1F	20	−	20	−
	END					

図 3.2 プログラムをアセンブルするステップ

BALR 命令は使用されていない．すなわち，このプログラムは，他のプログラムから最初の命令の番地をレジスタ15に設定した上で呼出されるものと考えられている（付録Aのサブルーチン結合の規約参照）．次にロード命令 L 1, FIVE がくる．表を探して命令記号Lに対するビット構成を求めることができる．Lに対するビット構成を機械語命令の適当な場所に入れる．次に FIVE の番地が必要である．しかし，この時点ではどこに FIVE があるかわからないのでその番地を設定することができない．インデックス・レジスタは使用されていないので，インデックス・レジスタ部に 0 を入れる．レジスタ15はベース・レジスタとして使用されていることがわかっているが，まだオフセットを計算することはできない．ベース・レジスタ15はこのプログラムの最初を指している．オフセットは FIVE の場所とプログラムの始まりの場所との差である．FIVE の場所は今はわからない．アセンブラは，処理している命令の相対番地を指す**ロケーション・カウンタ**の値を 4（ロード命令の長さ）だけ増す．

次の命令は加算命令である．命令コードは表で求まるが，FOUR に対する

オフセットはわからない．同様なことが格納命令でも起こる．DC 命令はデータを定義することを指令する擬似命令であり，FOUR に対しては“4”を作成する．最初の4バイト長の命令は相対番地0にある．また，次の2つの命令の長さは共に4バイトである．ロケーション・カウンタは先行する命令の長さによって増加してきており，この DC 命令を処理するときは値12を持っているので，この語は相対番地12に格納されることがわかる．記号“FOUR”は値12を持つという．次の命令はラベル FIVE を持ち，それに対するロケーション・カウンタの値は16である．ラベル TEMP は値20を持つ．このようにして**図3.2**の中央の列のコードが作成される．

ここでアセンブラは，プログラムの最初に戻って，**図3.2**の3列目に示したようにオフセットを記入することができる．

記号は定義より前に現われ得るので，（この例に示したように）入力を2回通すと処理上都合が良い．1回目は記号を定義するだけで，2回目に命令と番地を生成することができる．（1パス・アセンブラや多パス・アセンブラもある．これらの設計との関連についても本章で論じる）アセンブラの役割りは次のとおりである：

1. 命令を生成する．
   a. 命令欄の記号を調べ機械コードを作る．
   b. 他の欄の処理をする——各記号の値を求め，リテラルを処理し，番地を割当てる．
2. 擬似命令を処理する．

これらの仕事は2回の**パス**，すなわち，入力を繰り返し順番に走査すること，に分けることができる．各々の仕事については1つ以上のアセンブラ・モジュールが対応する．

パス1： 目的——記号とリテラルを定義する．

1. 機械命令の長さを決定する（MOTGET1）．
2. ロケーション・カウンタの値を決定する（LC）．
3. パス2まで記号の値を記憶しておく（STSTO）．
4. EQU，DS などの擬似命令の処理を行なう（POTGET1）．
5. リテラルを記憶する（LISTO）．

パス2： 目的——目的プログラムを生成する.

1. 記号の値を探し出す (STGET).
2. 命令を生成する (MOTGFT2).
3. データを生成する (DS, DC およびリテラルに対して).
4. 擬似命令を処理する (POTGET2).

図3.3および**図3.4**はパス1とパス2の処理の概要を示す. データ・ベースの特性やアルゴリズムの詳細は次節で展開する.

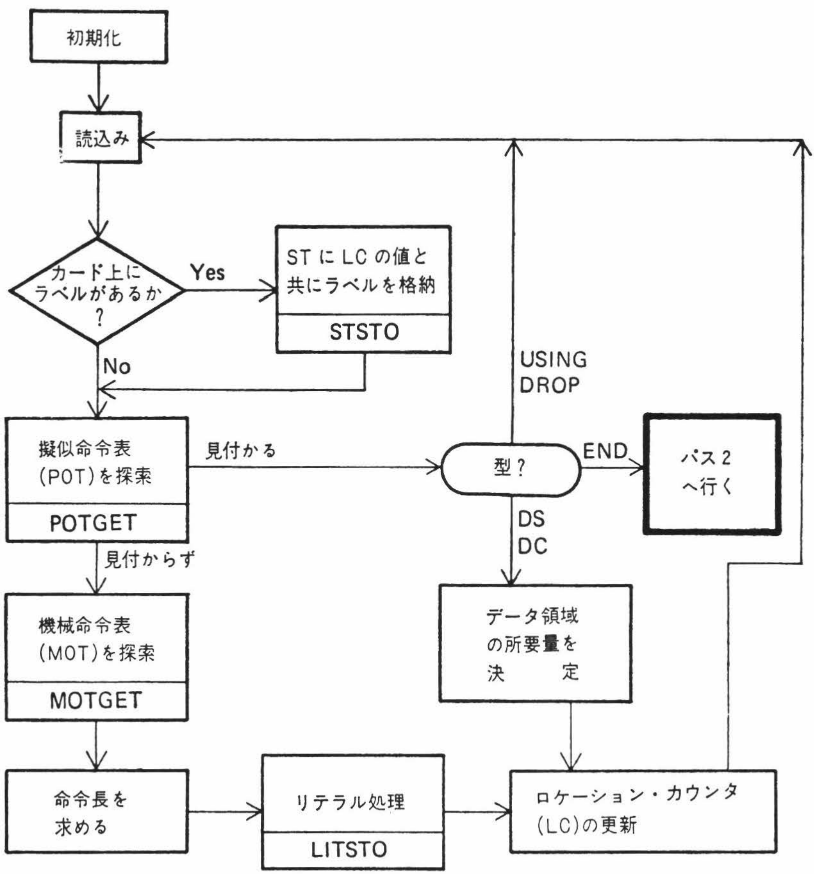


図 3.3 パス1の概略——記号の定義

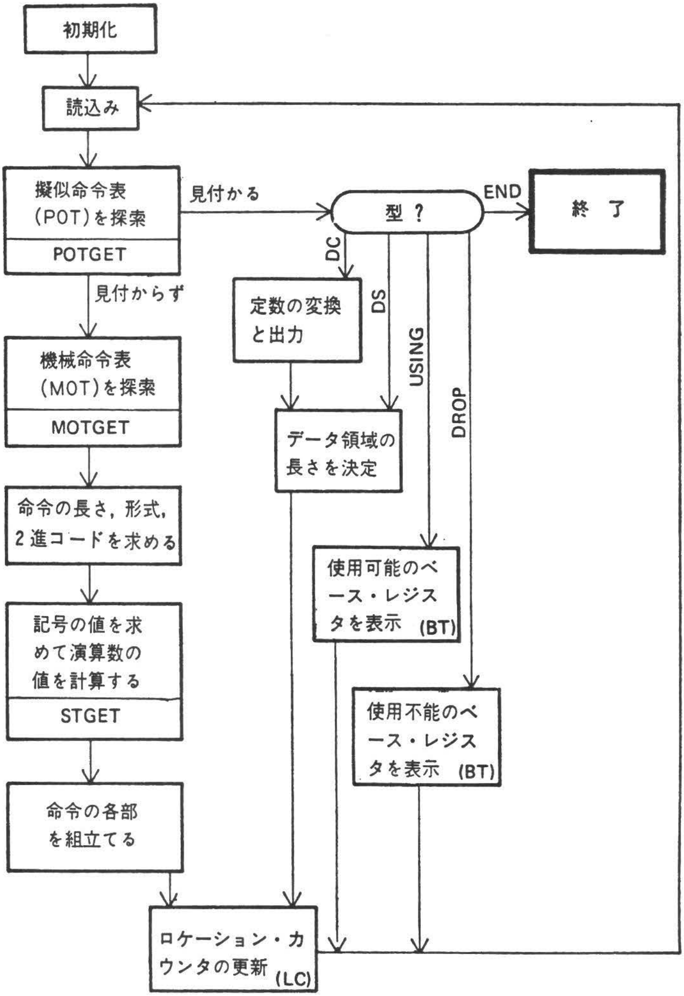


図 3.4　パス 2 の概略――各部の計算とコードの生成

3.2.2 データ構造

設計の第 2 段階はデータ・ベースを確立することである：

パス 1 のデータ・ベース：

1. 原始プログラム入力.

2. ロケーション・カウンタ (LC)——各命令の場所を示す.
3. 機械命令表 (MOT)——各命令の記号名と長さ（2，4または6バイト）を示す.
4. 擬似命令表 (POT)——各擬似命令の記号名とパス1で行なわれる処理を示す.
5. 記号表 (ST)——各ラベルと対応する値を格納する.
6. リテラル表 (LT)——出現した各リテラルとそれに割当てられた記憶場所を格納する.
7. 入力の写し——後のパス2で使用される. これは磁気テープやディスクまたはドラムなどの2次記憶に納めてもよいし，元の原始デックをアセンブラのパス2で再び読込んでもよい.

パス2のデータ・ベース：

1. パス1の原始プログラム入力の写し.
2. ロケーション・カウンタ (LC).
3. 機械命令表 (MOT)——各命令に対して次の項目を持つ：
   (a) 記号名，(b) 長さ，(c) 機械語の2進表現，(d) 形式（例えば，RS, RX, SI).
4. 擬似命令表 (POT)——各擬似命令に対する記号名とパス2での処理を示す.
5. 記号表 (ST)——パス1で作られ，各ラベルとその値を持つ.
6. ベース表 (BT)——どのレジスタが USING 擬似命令でベース・レジスタとして指定されているか，および，これらのレジスタの内容は何かを示す.
7. 作業領域 INST——命令の各部分（例えば，2進命令コード，レジスタ部，長さの指定部，変位部）が合成されるとき，その命令を納めておくために使用する.
8. 作業域 PRINT LINE——印刷結果を作成するために使用する.
9. 作業域 PUNCH CARD——ローダが要求する書式でアセンブルされた命令を出力するために使用する.

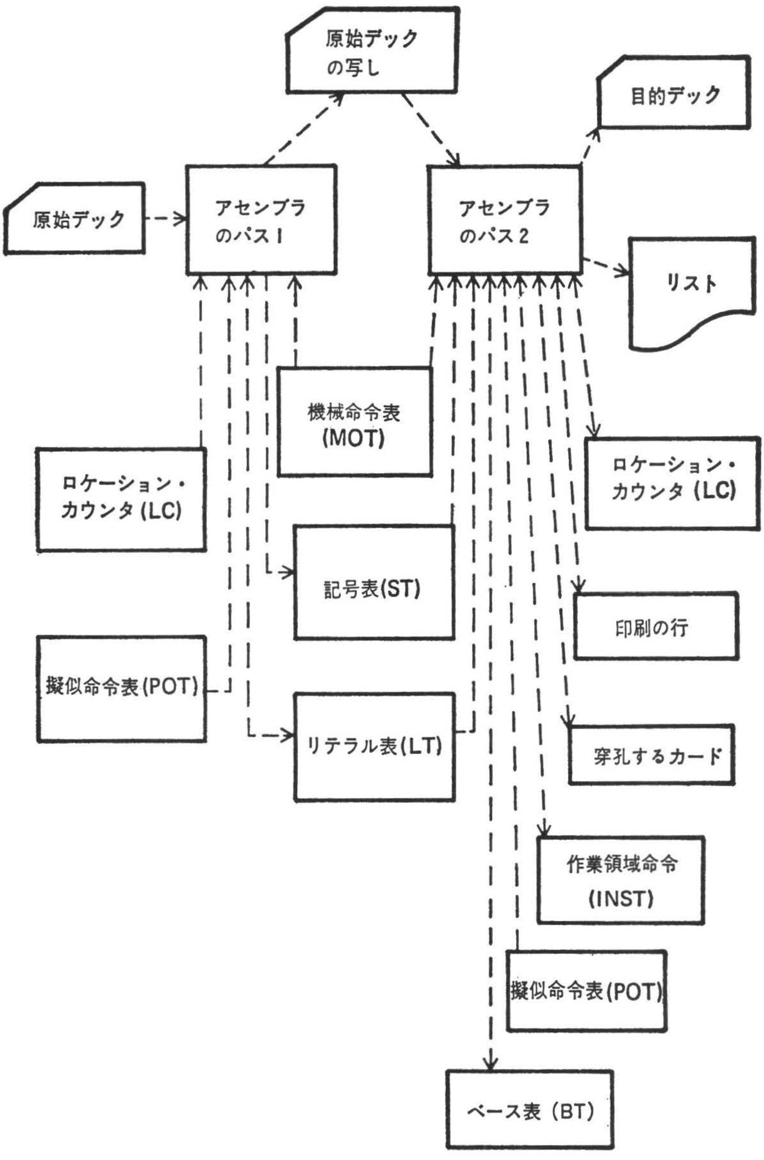


図 3.5 アセンブラのパスとデータ・ベースの関係

10. 出力デック――ローダが要求する書式でアセンブルされた命令.

図 3.5 はアセンブラのデータ・ベースと2つのパスの関係を示す.

3.2.3 データ・ベースの書式

設計の第3段階は各データ・ベースの書式と内容を規定することである. この仕事は, アセンブラの設計における特定のアルゴリズムの記述の前に行なう必要がある. アルゴリズム, データ・ベースおよびその書式はすべて関連しているので, 実際には, 設計者が使用しようとする書式とアルゴリズムの特性を心に描いて, すべての場合について条件が満されるまで繰り返すことによってその設計が行なわれる.

パス2は命令の名前, 長さ, 2進コードおよび形式を格納した機械命令表 (MOT) を必要とする. パス1では単に命令の名前と長さがわかればよい. このために別の書式と内容を持つ2つの独立な機械命令表を用いることもできるし, 2つのパスで同じ表を用いることもできる. 同様のことが擬似命令表 (POT) についてもいえる. また, 表の書式を一般化して MOT と POT とを1つの表にまとめることも可能である. ここでは1つの MOT と2つの POT を用いることにする.

どの情報がどのデータ・ベースに属するかが決定されると, 次に各欄の書式を規定することが必要になる. 例えば, どの書式で記号が格納されるのか (例えば, 左端をそろえ, 空白を詰め合せ, コードは EBCDIC または ASCII というふうに), また, コード化の規約は何かということである. 拡張2進化10進変換符号 (Extended Binary Coded Decimal Interchange Code : EDCDIC) は8ビット・バイトで文字を表現するための IBM 360 の標準コードである. 例えば, 文字 'A' は EBCDIC で2進の 1100 0001 (16進で C1) と表わされる.

機械命令表 (MOT) と擬似命令表 (POT) は**内容の固定した表**である. これらの表はアセンブル時に内容が変更されたり新たに記入されたりすることがない. **図 3.6** には機械命令表の書式と内容の例を示している. 記号名命令コードは表を引くための見出しである. その値は命令コードの2進表現であり, 機械コードを生成するために使用される. 命令長はロケーション・カウンタを更新するために使用され, 命令形式は機械語を形成するとき使用される.

各欄6バイト

記号命令コード (4-バイト) (文　字)	2進命令コード (1-バイト) (16　進)	命　令　長 (2-ビット) (2　進)	命令形式 (3-ビット) (2　進)	使用せず (3-ビット)
"A*bbb*"	5A	10	001	
"AH*bb*"	4A	10	001	
"AL*bb*"	5E	10	001	
"ALR*b*"	1E	01	000	
"AR*bb*"	1A	01	000	
...	...	...	...	
"MVC*b*"	D2	11	100	
...	...	...	...	

b、は"空白"を示す

コード：

命令長

01 = 1 半　語　= 2 バイト
10 = 2 半　語　= 4 バイト
11 = 3 半　語　= 6 バイト

命令形式

000 = RR
001 = RX
010 = RS
011 = SI
100 = SS

図 3.6　パス1とパス2の機械命令表 (MOT)

各欄8 バイト

擬似命令 (5-バイト) (文　字)	擬似命令表を処理するルーチンの番地 (3-バイト＝24ビットの番地)
"DROP*b*"	P1DROP
"END*bb*"	P1END
"EQU*bb*"	P1EQU
"START"	P1START
"USING"	P1USING

：これらはパス1のルーチンのラベルである。実際には表に物理的な番地が記入されている。

図 3.7　パス1の擬似命令表 (POT)（パス2も同様）

←　各欄14バイト　→

記　号 (8-バイト) (文　字)	値 (4-バイト) (16進)	長　さ (1-バイト) (16進)	再配置 (1-バイト) (文　字)
"JOHN*bbbb*"	0000	01	"R"
"FOUR*bbbb*"	000C	04	"R"
"FIVE*bbbb*"	0010	04	"R"
"TEMP*bbbb*"	0014	04	"R"

図 3.8　パス1とパス2の記号表 (ST)

図 3.7 は擬似命令表の例を示している．各擬似命令は，これを処理するためのアセンブラ・ルーチンへのポインタと共に記入されている．

記号表とリテラル表（**図 3.8**）は各記号に対して，名前とアセンブル時の値の欄の他に，長さの欄と相対番地表示欄を持っている．長さの欄はこの記号が付けられた命令またはデータのバイト長を示している．次の例を考えよう：

```
COMMA    DC    C','
F        DS    F
AD       A     1,F
WORD     DC    3F'6'
```

記号 COMMA は長さ1，Fは長さ4，AD は長さ4（4バイト長の加算命令)，WORD は長さ4（係数3は長さを定めるのに無関係）である．記号 *（ロケーション・カウンタの現在の値)はつねに長さ1とされる．記号が(EQUによって）他の記号と等しい値を持つと定義されたときは，長さはその記号の持つ長さに等しくされる．長さの欄は，ある種の SS 型命令において必要とされる長さの演算数を計算するためにアセンブラで使用される．

相対番地表示欄は，記号の値が絶対値である（プログラムがコア中で移動しても変更しない）か，プログラムの最初に対する相対値であるかをアセンブラに教える．アセンブラは簡単な方法により，1つの記号がどちらに属するかを決定することができる．記号が定数（例えば6）や絶対記号に等しい値を持つと定義されていれば，この記号は絶対値を持つ．それ以外は相対値である．記号表の相対番地表示欄には，記号が相対値のとき‘R’が，絶対値のとき‘A’が記入される．実際のアセンブラでは一般的にもっと複雑なアルゴリズムが使用されている．

各欄4バイト

	使用可標示子 (1-バイト) (文 字)	相対番地 ベース・レジスタの内容 (3-バイト=24-ビットの番地) (16進)
1	"N"	–
2	"N"	–
⋮	⋮	
14	"N"	–
15	"Y"	00 00 00

15欄

コード＝
使用の可否
Y～ USING擬似命令で指定されたレジスタ
N～ USING擬似命令では指定されたことがないか，
または，擬似命令で使用不可とされたレジスタ

図 3.9 パス2のベース表 (BT)

図3.9は，適当なベース・レジスタを選択し，それによって正しいオフセットを計算するためにアセンブラによって使用されるベース表の例を示している．基本的に，アセンブラは番地（オフセット，ベース・レジスタ番号，インデックス・レジスタ番号）をほとんどすべての記号名による参照に対して生成しなければならない．記号表は，記号がプログラムの先頭に対して持つ相対的な番地を納めている．番地を生成するとき，アセンブラはベース表を使用して，参照している記号に最も近い値を持つベース・レジスタを選ぶ．次に番地が形成される：

ベース・レジスタ番号＝参照している記号に最も近い値を持つベース・レジスタ．

オフセット＝（記号表の中での記号の値）－（ベース・レジスタの内容）．

次に示したアセンブラ語のプログラムを用いて**可変の表**（記号表，リテラル表およびベース表）の使用法を説明し，次節に示されるアルゴリズムを生み出した動機を説明する．われわれはこのプログラムをアセンブルすることのみに関心があり，プログラムが何をするかということは考えない．

アセンブラのパス1の目的（記号とリテラルの定義）にしたがって，次に示す記号とリテラルの表を作成することができる．

アセンブラ原始プログラムの例

```
文番号
 1    PRGAM2     START   0
 2               USING   *,15
 3               LA      15,SETUP
 4               SR      TOTAL,TOTAL
 5    AC         EQU     2
 6    INDEX      EQU     3
 7    TOTAL      EQU     4
 8    DATABASE   EQU     13
 9    SETUP      EQU     *
10               USING   SETUP,15
11               L       DATABASE,=A(DATA1)
12               USING   DATAAREA,DATABASE
13               SR      INDEX,INDEX
14    LOOP       L       AC,DATA1(INDEX)
15               AR      TOTAL,AC
16               A       AC,=F'5'
17               ST      AC,SAVE(INDEX)
18               A       INDEX,=F'4'
19               C       INDEX,=F'8000'
20               BNE     LOOP
21               LR      1,TOTAL
22               BR      14
23               LTORG
24    SAVE       DS      2000F
25    DATAAREA   EQU     *
26    DATA1      DC      F'25,26,97,101 ....'
                              [2000 個の数 ]
27               END
```

アセンブラは**図 3.3** 流れ図のように，上述のプログラムをロケーション・カウンタの更新を行ないながら走査する．そして原始プログラムのラベル部に使用されている各記号に対して，記号表の中に1つずつ欄を作り，記号と値を記入する．最初の記号 PRGM2 に対する値は相対番地である． IBM の規約により長さは1である．

変数表

記号表

記号	値	長さ	再配置
PRGAM2	0	1	R
AC	2	1	A
INDEX	3	1	A
TOTAL	4	1	A
DATABASE	13	1	A
SETUP	6	1	R
LOOP	12	4	R
SAVE	64	4	R
DATAAREA	8064	1	R
DATA1	8064	4	R

リテラル表

A(DATA1)	48	4	R
F'5'	52	4	R
F'4'	56	4	R
F'8000'	60	4	R

次の LA 命令は4バイト長，SR は2バイト長であるから，これに基づいてロケーション・カウンタが更新される．引続いて，5つの記号が EQU 擬似命令で定義されている．これらの記号は EQU 文の引数部に与えられた値と共に記号表に登録される．さらに，L 命令は4バイト，SR 命令は2バイトであることがわかり，ロケーション・カウンタが更新される．（これまでの擬似命令は目的コードを生成しないのでロケーション・カウンタの値に影響しないことに注意.）このようにして LOOP が出てきたときにロケーション・カウンタは値12を持つ．したがって，LOOP は記号表に値12と共に記入される．この記号は再配置可の変数であり，このことも表に示される．4バイト命令に付けられた記号であるので長さは4である．他のすべての記号も同様にして登録される．

同じパスにおいてすべてのリテラルも検出され，リテラル表に登録される．最初のリテラルは文11に現われ，その値はそのリテラルを納めている場所の番地である．このリテラルは最初に現われたものであるから，リテラル領域の最初の番地を与えられる． LTORG 擬似命令（文23）はその場所でリテラル表を生成する．ロケーション・カウンタは次の2語境界に更新され48となる．し

たがって '=A(DATA)' の値はそれの置かれる番地，すなわち，48である．同様にしてリテラル F'5' の値はリテラル表の次の番地，すなわち，52とされ，以下同様に割当てられる．

リテラル表と記号表が完成すると，パス2を開始できる（**図3.4**）．パス2の目的は引数を演算してコードを生成することである．命令に適当な番地部を作るためにはベース・レジスタがわからなければならない．オフセットを計算するためにはベース・レジスタの内容がわからなければならない．もちろん，アセンブラは実行時におけるベース・レジスタの値を知らないが，プログラムの先頭からの相対的な値は知っている．したがって，アセンブラは“内容”として相対的な値を記入する．この値がオフセットを計算するのに用いられる．

USING 擬似命令を処理すると次のベース表が作成される：

ベース表（使用されるベース・レジスタのみ示す）

1) 文2の実行後

ベース	内容
15	0

2) 文10の実行後

15	6

3) 文12の実行後

13	8064
15	6

パス2の各命令に対して下に示したのと等価な機械語の命令が作られる．例えば，文3に対しては，

1. 記号表で SETUP を探す（6である）．
2. 機械命令表で命令コードの値を求める（Lに対する2進コード）．
3. 番地を形成する．
   a. ベース・レジスタを決定する——SETUP に最も近い値を持つベース・レジスタを選ぶ（レジスタ15）．
   b. オフセット＝記号の値－ベース・レジスタの内容＝6－0＝6.
   c. 番地を形成する——オフセット（インデックス・レジスタ，ベース・レジスタ）＝6(0,15).

4. 適当な形に出力コードを作る.

同様にして，次に示したように残りのコードに対して命令を生成する.

生成された"機械"コード

対応する文番号	番地	命令/データ	
3	0	LA	15,6 (0,15)
4	4	SR	4,4
11	6	L	13,42 (0,15)
13	10	SR	3,3
14	12	L	2,0 (3,13)
15	16	AR	4.2
16	18	A	2,46 (0,15)
17	22	ST	2,58 (3,15)
18	26	A	3,50 (0,15)
19	30	C	3,54 (0,15)
20	34	BC	7,6 (0,15)
21	38	LR	1,4
22	40	BCR	15,14
23	48	8064	
	52	X'00000005'	
	56	X'00000004'	
	60	8000	
24	64	·	
	·	·	
	·	·	
25	8064	X'00000019'	
		·	
		·	
		·	

3.2.4 アルゴリズム

図 3.10 および**図 3.11** の流れ図は IBM 360 計算機用アセンブラのアルゴリズムを比較的詳しく示したものである．これらの図は複雑なアセンブラで行なわれる処理を簡略化して示しているが，論理的な処理の大半は含まれている．

パス 1：記号の定義

パス 1 の目的は，各命令とデータを定義する擬似命令とに番地を割当て，原始プログラムのラベル部に現われる記号に対して値を定義することである．最初ロケーション・カウンタ (LC) はプログラムの先頭（相対値 0）に設定され

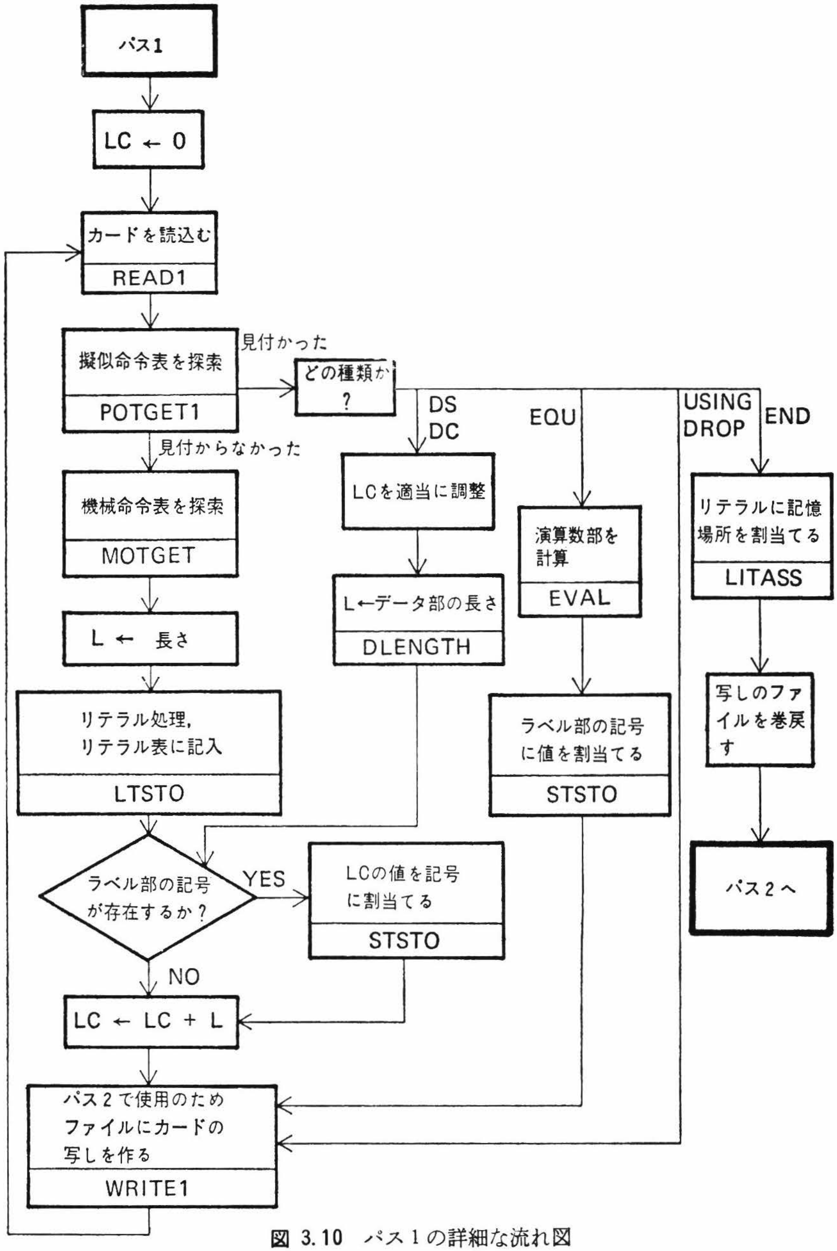


図 3.10 パス1の詳細な流れ図

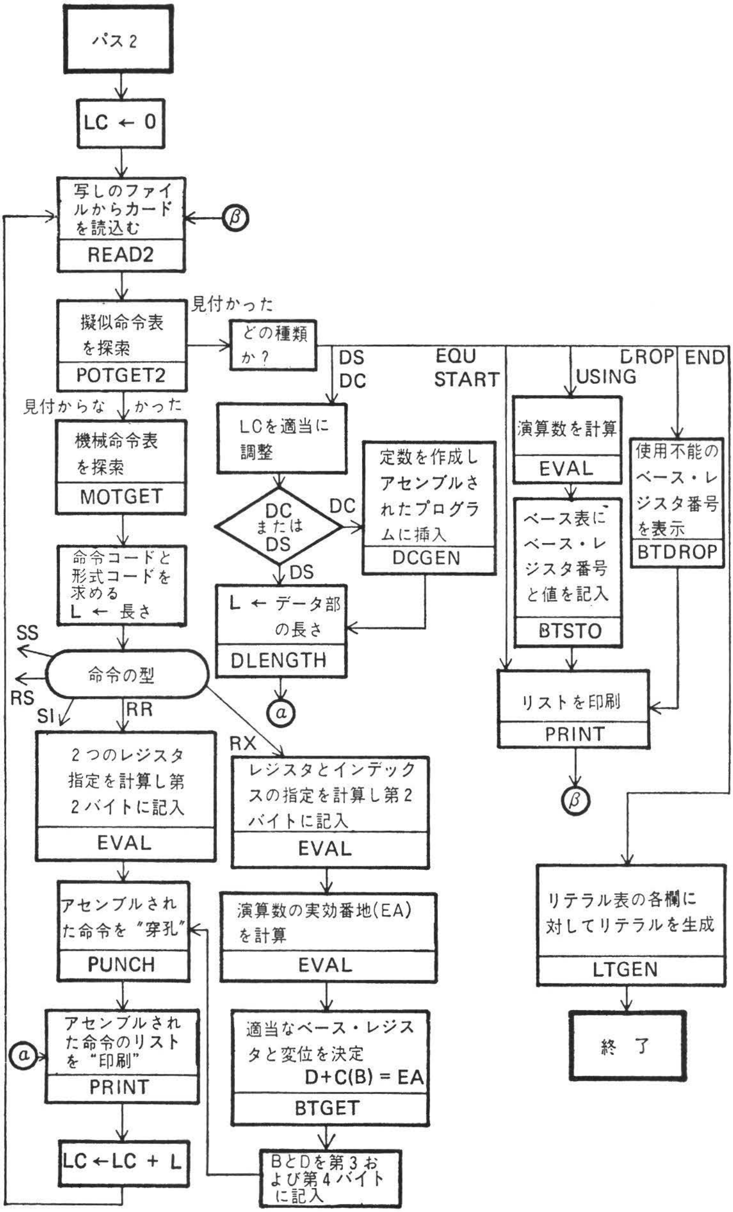


図 3.11 パス2の詳細な流れ図

る．次に原始文が読込まれ，命令コード部が擬似命令であるかどうか調べられる．擬似命令でなければ機械命令表(MOT)を探して原始文の命令コード部と一致するものを求める．求めた MOT の欄は命令長（2，4または6バイト）を指定する．演算数部にはリテラルがあるかどうか調べられる．新しいリテラルが見付かると，後で処理するために，リテラル表(LT)に登録される．次に原始文のラベル部に記号があるかどうか調べて，記号があれば記号表にロケーション・カウンタの現在値と共に記入される．最後に，ロケーション・カウンタの値が命令の長さだけ増加され，パス2で使用するために原始カードの写しを作る．これまでの手順は以下の命令についても繰り返される．

今述べたループは，最も重要な部分であるけれども，物理的にはパス1の小さな部分を占めるにすぎない．パス1およびパス2の最も大きな部分は各種の擬似命令に対して必要な特別な処理に当てられている．**図3.10**には2，3の代表的な擬似命令の処理だけが示されている．他の擬似命令の処理はきわめて明解である．

さて，擬似命令の処理では何をしなければならないかを考えてみよう．USING および DROP に対しては最も単純な手続きが用いられる．パス1では記号を定義したりロケーション・カウンタに影響を与える擬似命令のみが処理される．USING や DROP はこれに関係がない．アセンブラはパス2のために USING や DROP カードを保存するだけでよい．

EQU 擬似命令の場合はパス1でラベル部の記号を定義するだけでよい．このためには演算数部の式を計算しなければならない．（EQU 文の演算数部の中の記号はこの文以前に定義されていなければならない．）

DS および DC 擬似命令は，パス1でロケーション・カウンタと記号の定義の両方に影響する．要求される記憶容量のバイト数を決定するために演算数部を調べる必要がある．ある種の整合条件（例えば，語は4バイトの倍数から始まらなければならない）によって，記号を定義する前にロケーション・カウンタを調整しなければならないことがある．

END 擬似命令が出てくるとパス1は終了する．パス2に制御を移す前に行なっておかなければならない“雑用”がある．これらにはパス1の間で集められたリテラルに記憶場所を割当てることがある．これは DC 擬似命令の処理に

類似している．また，パス2の処理のために初期条件が設定される．

パス2：コードの生成

パス1ですべての記号が定義されると，各カードを処理し，命令コードと演算数部の値を決定してアセンブルを完了することができる．さらに，パス2では生成したコードを後でローダが処理できるような書式に構成し，原始プログラムと生成されたバイトの16進表現とを含めてアセンブル結果の印刷を行なう．ロケーション・カウンタはパス1と同様に初期化され，次のように処理が続けられる．

パス1で作られた原始プログラム・ファイルからカードを読込む．パス1と同様に，擬似命令かどうか命令コード部を調べる．擬似命令でなければ機械命令表（MOT）を探してカードの命令コード部と一致しているものを求める．一致した MOT の欄には命令の長さ，2進コードおよび形式が示されている．命令の形式が異なると演算数部の処理も幾らか異なる．

RR 型命令に対しては2つのレジスタ指定部の値が計算される．この計算は，

```
AR        2,3
```

の場合のように非常に簡単なときもあれば，

```
MR        EVEN,EVEN+1
```

と少し複雑なときもある．この2つの部分は，2バイト長の RR 型命令において2番目のバイト中の対応した4ビットの部分に挿入される．

4バイト長の RX 型命令に対しては，RR 型命令の場合のレジスタ指定と同様にして，レジスタおよびインデックス部が計算されて処理される．記憶番地部が計算され，実効番地(Effective Address：EA)が作られる．ベース・レジスタ表（BT）を調べて $D=EA-C(B)<4096$ であるような適当なベース・レジスタ（B）を求めなければならない．ベース・レジスタが定まると変位部が計算できる．次に，4ビットのベース・レジスタ指定と12ビットの変位部は命令の3番目および4番目のバイトの中で組立てられる．流れ図（**図3.11**）には

RR 型および RX 型命令のみを示してある．他の型の命令も同様に処理される．

命令が組立てられたら，後でローダが処理するために必要な書式にして出力される．普通，幾つかの命令が1枚のカードに納められる（詳細な議論は第5章を参照）．次に原始カードの写し，割当てられた記憶場所および目的コードの16進表現を含んだ1行が印刷される．最後に，ロケーション・カウンタが増加され，処理は次のカードへと続けられる．

パス1と同様にパス2でも擬似命令は特別の処理を必要とする．EQU 擬似命令は，記号定義をパス1で終了しているので，パス2ではほとんど処理することがない．EQU カードを印刷することだけが必要である．

USING および DROP 擬似命令は，パス1ではほとんど無視されたが，パス2では幾らかの処理が必要である．擬似命令の演算数部の値が計算され，USING のときは対応したベース表の欄が使用可と表示され，DROP のときには使用不可と表示される．パス2においてベース表は，記憶装置を参照する機械命令のベースを選び変位部を計算するのに活用される．

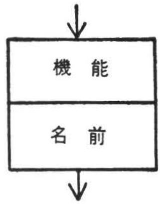
DS および DC 擬似命令は基本的にパス1と同様に処理される．しかしパス2では DC 擬似命令に対して実際のコードを生成しなければならない．指定されたデータの型によっては各種の変換（例えば，浮動小数点2進数から文字表現）および記号の値を求めること（例えば，番地定数）が必要である．

END 擬似命令は原始プログラムの終りを示しアセンブルを終結させる．このとき各種の“雑用”を行なう必要がある．例えば，リテラル表の中に残っているリテラルに対してコードを生成することなど．

3.2.5 機能単位を探すこと

これまでの設計を振り返って，分割できる機能を求めてみよう．
これらの機能は (1)汎用のものと (2)特定のものとの2種に分類できる．

パス1の流れ図（**図3.10**）とパス2の流れ図（**図3.11**）で，論理的に独立な部分を調べる．分割できそうな部分は流れ図で次のような形で示されている．ここで“名前”は機能に付けられた名前である（例えば，MOTGET，EVAL，PRINT）：



下には２つのパスの中の分割可能な機能の幾つかを示してある：

パス１：

1. READ1 ——次のアセンブラ原始カードを読込む.
2. POTGET1 ——原始カードの命令欄と一致するものをパス１の擬似命令表（POT）で**探す**.
3. MOTGET1 ——原始カードの機械命令欄と一致するものをパス１の機械命令表（MOT）で**探す**.
4. STSTO ——ラベルとそれに対する値を記号表（ST）に**登録する**. 既に表に記号があれば誤り表示を行なう（記号の多重定義).
5. LTSTO ——リテラルをリテラル表（LT）に**登録する**. 同じリテラルは２回登録しない.
6. WRITE1 ——パス２で使用するために原始カードの写しを記憶装置に作る.
7. DLENGTH ——DS または DC 擬似命令の演算数欄を調べ所要の記憶容量を**決定する**.
8. EVAL ——定数と記号から成る算術式（例えば, 6, ALPHA, BETA＋4＊GAMMA）の**値を求める**.
9. STGET ——特定の記号を記号表（ST）の中で**探す**（STSTO および EVAL にて使用される).
10. LITASS ——リテラル表の中のリテラルに記憶場所を割当てる（DLENGTH を使用することがある).

パス２：

1. READ2 ――ファイルの写しからアセンブラ原始カードを読込む.
2. POTGET2 ――POTGET1 と同じ（POT を探す）.
3. MOTGET2 ――パス1に同じ（MOT を探す）.
4. EVAL ――パス1に同じ（式を演算する）.
5. PUNCH ――生成された命令をカードの書式に変換し，データが一杯になればカードをせん孔する.
6. PRINT ――相対番地と生成されたコードを文字形式に変換し，原始カードの写しと共に印刷する.
7. DCGEN ――DC 擬似命令の処理を行ない，目的コードを生成する（EVAL および PUNCH を使用する）.
8. DLENGTH ――パス1に同じ.
9. BTSTO ――ベース表（BT）の適当な欄にデータを挿入する.
10. BTDROP ――BT の適当な欄に“使用不可”の表示子を挿入する.
11. BTGET ――使用可能なベース・レジスタをベース表（BT）で探して，実効番地をベースおよび変位に変換する.
12. LTGEN ――リテラルにコードを生成する（DCGEN）を使用する.

これらの機能は設計の過程全体（問題の設定，データ・ベース，アルゴリズム，機能単位など）を通じて独立に吟味されなければならない．これらの機能は独立の外部サブルーチン，内部サブルーチン，またはパス1およびパス2のプログラム・セクションなど実際に適した形で組込むことができる．いずれにせよ，機能を分割して取り扱えることはアセンブラの構造と各部分の構造の設計を非常に容易にする．アセンブラを1つのプログラム（1,000ないし 10,000の原始文）として考えるより，比較的小さな，あまり複雑でないルーチンの集合として考えることができる．

これら機能ルーチンの論理はきわめて容易に理解できるから，ここではすべてのルーチンを調べることはしないが，これらからは2つの興味あることが見出される：(1)幾つかのルーチンは値の計算を行なう（例えば，DLENGTH, EVAL, DCGEN）．(2)他の幾つかのルーチンは表を分類したり探索したりする処理を含む（例えば， POTGET1, POTGET2, MOTGET1, MOTGET2,

LTSTO, STSTO, STGET). コンパイラの組込みを扱っている第8章で構文解析と算術式計算を論じる．これらの多くはアセンブラの機能部分として適用可能である．

アセンブラ組込みに関して論じられた表の処理は，多くの応用プログラムはもとより，コンパイラ，ローダ，ファイル・システムおよびオペレーティング・システムなどを含むほとんどすべてのシステム・プログラムに使用されている．

データ構造とデータ編成の処理はシステム・プログラミングに重要な役割りを果している．表の欄に記入したり欄を探したりすることはアセンブル時間の最も大きい部分を占めることが多いので，次節ではこれらの処理の技法を調べてみよう．

3.3　表の処理　：　探索と分類

設　問

多量の情報を含む表から要求された項目を迅速に，かつ，容易に取り出したり記入したりできるように表を整理しておくことがしばしば必要となる．ここではアセンブラ・プログラムで使用される**記号表**のように，見出し語を基準として表の欄が構成される場合に限定して考えることにする．

アセンブラの記号表は数語から成る固定書式の欄から構成されている．表には記号名，その値および再配置可などの各種の属性が記入される．記号名は**見出し**であり，探索のときに一致するかどうかを調べるための記号列である．各記号はそれに対応した**値**を持つ．（電話番号簿で加入者名が見出しであり，電話番号が値であるのと同じである．）アセンブラの記号表には2つの注意すべき点がある．

1. 記号は収録された順に表に記録されており，整理して並べられていない．
2. 記号とそれに対応したデータは表の連続した場所に置かれている．これらは表の一端から始めて，すき間を空けずに詰め合せてある．

これら2つは，1つずつ欄を作り，欄の場所を計算せずに直ちに記入するような大半の表についても当てはまる．

表の探索

探索の問題は次のとおりである．見出し語を与えられ，表の中でそれに一致する欄を探し，その値を返却する．特別な問題は，同じ見出し語が2個以上存在する場合と全く存在しない場合であり，表の機能にしたがって個別に処理する必要がある．アセンブラの記号表でこれらの特別な場合は多重定義の記号と未定義記号に対応する．

3.3.1 逐次探索法

分類その他の方法で欄が順に並べられていない表の場合には，与えられた見出し語を探すのに表のすべての欄を比較する方法がある．この方法は**逐次探索法**（linear search）として知られ，**図3.12**に例が示されている．例で記号とその値は，DS で定義された SYMTBL という名前の配列の中で，連続して格納されている．語 LAST は“表の終り”の場所を納めている．

```
            LA      4, SYMTBL        表の開始点
 LOOP       CLC     0(8,4), SYMBOL   記号の比較
            BE      SYMFOUND         等しい
            A       4,=F'14'         次の記号に移る
            C       4,LAST           表の終りか
            BNE     LOOP             終りでなければLOOPに戻る
NOTFOUND            (記号は見付からなかった)
              ⋮
SYMFOUND            (記号が見付かった)
              ⋮
SYMBOL      DS      CL14             探索すべき記号,
                                     長さ14の文字列
SYMTBL      DS      100CL14          記号表の領域（各欄14バイト）
LAST        DC      A(---)           記号表の終りの番地
```

図 3.12 逐次探索プログラムの例

例のループは見出し語（記憶場所 SYMBOL に入っている）と表の各項目を順に比較する．一致するものが見付かれば SYMFOUND へ行く．一致するものが見付からなければ NOTFOUND へ行く．

欄を見付けるまでに逐次探索法では平均的に表の半分を探すことになる．したがって，欄を探す時間の平均値は

$$T(avg)=[1つの欄を探す時間]\times\frac{N}{2}$$

このような逐次探索の手続きは表が短いとき速く，また単純であるが，長い表のときは非常に遅い．これは，項目が順番に並べてない辞書を引くのと同様であり，平均的に辞書の**半分**を探さなければならないということを知ればいやになるであろう．

3.3.2 2分探索法

われわれが辞書を引くとき単語を1ページずつ探すことはしない．辞書の中で単語のありそうな場所（ページ）の大体の見当を付けて，そのページを開く．そのページになければ適当なページ数だけ右または左へ行ってもう一度調べる．辞書は順番に並べてあり（BはAの次とかSはGより後であるとか），その法則を知っているのでどちらへ行けばよいかは分っている．このような文字の並べ方を**辞引式順番**（lexicographical order）という．

並べられた表を探索するもっと規則的な方法は表の中央から探索を始め，見出し語を中央の欄と比較することである．見出し語は比較した項目に等しいが，より大きいか，より小さいかである．これらの各々にしたがって次のように進む：

1. 等しければその記号は見付けられた．
2. より小であれば表の上半分を新しい表として探す．
3. より大であれば表の下半分を使用する．

この方法は各判定で表を半分に割り，規則的に探索すべき項目をくくり出す．探索は，表の長さが1となり，しかも項目が見付からなかったときに，‘項目なし’の状態で終了する．

この探索法の例として15項目の表を考えよう（**図3.13**）．
例えば，項目 IF を探すことにする（簡単のために値は省略してある）．まず，IF と中央の項目 LO を比較し， IF は表の上半分になければならないことがわかる．2回目は上半分の表の中央の項目 FU と比較し IF は4つに割った2番目の表にあることがわかる．3回目の IW との比較で IF は8つに割った3

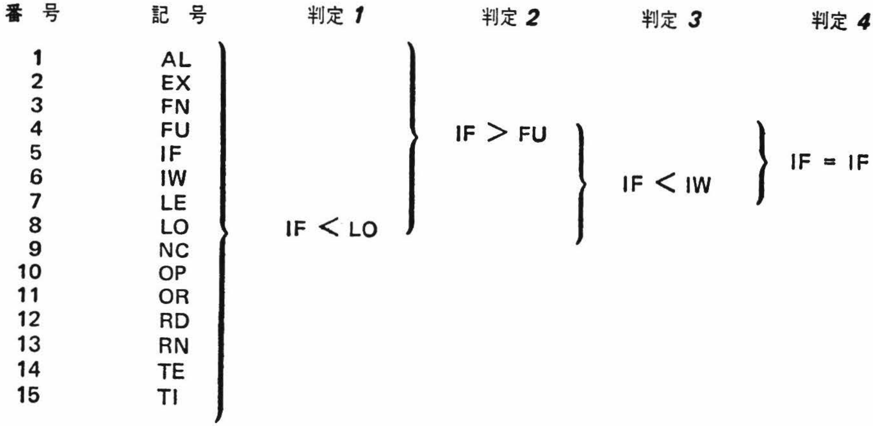


図 3.13　2分探索法

番目の表（すなわち，項目4から6）にあることがわかる．そして最後に5番目の項目と比較される．4回目の比較で項目が見付からなければ表には存在しないことになる．

このような表のくくり出しの方法は，原理は明解であるがその組込みはやや複雑である．この方法は**2分探索法**または**対数探索法**として知られており，各判定によって表が2分されるので，最大 $\log_2(N)$ 回の判定が必要であることがわかる．

逐次探索法と2分探索法と比較すると，AおよびBを各々の方法の1回当りの判定時間とすると，

$$T(\text{逐次法}) = A*N$$
$$T(2\text{分法}) = B*\log_2(N)$$

2分探索法はより複雑であるので，定数BはAよりかなり大きいと考えられる．したがって，TとNの関係は2つの探索法において**図3.14**のようになる．

Nが小さなときには逐次探索法を使用し，Nが大きいときは2分探索法を使用すべきである． IBM 360 のような計算機では曲線の交点は50～100項目の当りである．この数は，他の計算機ではハードウェアによって10から1000の範囲で変化する．

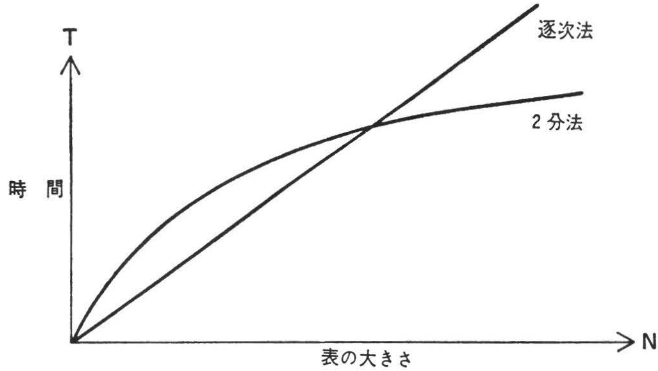


図 3.14　探索時間と表の大きさの関係

図 3.15 は2分探索プログラムの例を示している．2分割の処理を行なうことから，効率と簡単のために表の大きさを2の幅乗（例えば，(2, 4, 8, 16……) と仮定している．この条件は表の終りに“ダミー”の欄（例えば，記号の場合 ZZZZ ZZZZ）を加えておくだけでよい．

```
          L      5,LAST           表の大きさを設定 (2N*14バイト)
          SRL    5,1              シフトして2で除す
          LR     6,5              レジスタ6に移す
LOOP      SRL    6,1              再び表の大きさを2分する
          LA     4,SYMTBL(5)      表の欄の番地を設定
          CLC    0(8,4),SYMBOL    記号と比較
          BE     FOUND            記号が一致，欄が見付かる
          BH     TOOHIGH          SYMTBLの欄>SYMBOL
TOOLOW    AR     5,6              表の上の方へ移る
          B      TESTEND
TOOHIGH   SR     5,6              表の下の方へ移る
TESTEND   LTR    6,6              終了の判定
          BNZ    LOOP             否，次の欄を判定せよ
NOTFOUND  (記号は見付からなかった)
                 ⋮
FOUND     (記号が見付かった)
```

図 3.15　2分探索プログラムの例

3.3.3　分　類　法

ある種の目的には逐次探索法より2分探索法の方の効率が良いことは明らか

であるが，この探索法には並べられた表が必要である．このような表はそれ程簡単には作れない．アセンブラの機械命令表（MOT）および擬似命令表（POT）は固定の表であり，はじめに手細工で並べておくことができる．しかしながら，普通，表は順番に並べられた形で作られるのではない．実際アセンブラによって作成される記号表においては，記号はラベル部に現われた順に表に登録されるので，通常並べられているとは到底いえない．

3.3.3.1　互換分類法

ここでは表を分類する問題を考える．表を分類する方法には簡単なものから複雑なものまであるが，**図 3.16**は**互換分類法**（**interchangeable** sort, **bubble** sort, **sinking** sort, **sifting** sort などともいわれる）による分類を行なうプログラムの部分を示している．この簡単な分類法は表の隣り合った項目を順番に置換え（互換）る．このような分類法のアルゴリズムはあまり効率が良いとはいえないが簡単である．例によってこれがどのように働くかを見てみよう．**図 3.17** に示した12個の数を考える．図の各列は，隣り合った数字で順番に並んでいないものを置換えるパスを示している．この表の場合には7回のパスでデータの分類が完了している．最悪の場合にはN−1 パス（この場合には11パス）必要となる．このように互換分類法では既に表の中にある項目間の順序付けが

```
         L       5,LAST
         LA      4,SYMTBL
LOOP     CLC     0(8,4),14(4)       隣りの記号と比較－8バイト
         BNH     OK                 正しい順番か
         MVC     TEMP(14),0(4)      欄の入換え
         MVC     0(14,4),14(4)      . . .
         MVC     14(14,4),TEMP      . . .
OK       A       4,=F'14'           次の欄に移る
         C       4,LAST             最後の欄か
         BNE     LOOP               否
           ⋮
SYMTBL   DS      0F                 記号表
         DS      100CL14            各欄14バイト
TEMP     DS      CL14               臨時使用の欄
LAST     DC      A(----)            表中の次の空欄の番地
```

図 3.16　互換分類プログラムの例

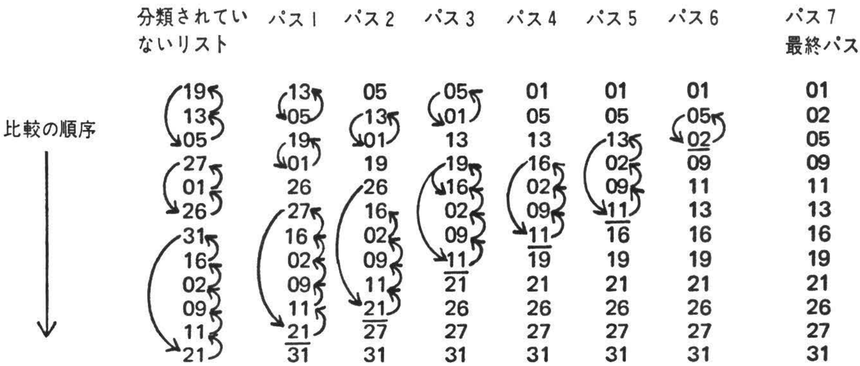


図 3.17　互換分類法

利用できる．また，各パスによって少なくとも1項目のデータが完全に並べられて表の最後に置かれる（この場合最初31，次に26というように)．したがって，(1)各パスで分類する表の長さを短かくして行く，(2)完了したかどうかを判定する，ことによってこの分類法はより効率良く実行することができる．このように最適化された分類法は約 N*(N−1)/2 回の比較を要し，時間はほぼ N^2 に比例する．

通常もっと時間の短い能率の良い分類法が要求されるので，種々の分類法が考えられている．分類法は次の3つの型に分類できる：(1) 1数字を判定する**分配法**(distributive sort)，(2) 2つの見出し語を比較する**比較法**，および，(3)見出し語を，その記号が置かれる位置の近くの番地に変換する**番地計算法**．

3.3.3.2 Shell 分類法

高速の比較法のアルゴリズムの1つに D. L. Shell の考案した（第10章の参考文献参照） **Shell 分類法**がある． この方法の効率は比較法に対する最適値に近い． Shell 分類法はデータ項目を交換するという点で互換法に類似している． しかし， この方法では比較を ‘d’ だけ離れた項目と比較する点が異なっている．この方法では本来の場所から離れて置かれている項目が互換分類法よりずっと速く移動することになる．項目の比較はパス毎に d の値を小さくし

て繰り返される．通常つぎのようなdが用いられる．

$$d_{i+1}=\frac{d_i+1}{2}$$

各項目はdだけ先にある項目と比較され，先の項目の値が小さいときは置換えが行なわれる．分類は次の項目に移ってdだけ先の項目（存在すれば）との比較が続行される．置換えが必要であれば実行し，比較が続けられる．この処理は1つのdの値について置換えが起こらなくなるまで繰り返される．この処理は**泡出し**（bubbling）と呼ばれる．これは値の小さい項目が上へ浮び上って行く過程が水槽の中の泡に似ていることからきている．1つの値dについて泡が出なくなると，dの値を小さくしてこの処理が続けられる．

1回のパスの効果が他にどのように及ぶかを示すことが難かしいので，Shell分類法の処理時間を予測することは困難である．d=1の泡出しは分類を完結させるので，上述のdの値の計算法が使用されるとき，パスの個数は約 $\log_2(d)$ であることは明らかである．経験的に Shell 分類法の所要時間はN項目のとき約 $B*N*(\log_2 N)^2$ である．定数Bは比較的小さい．したがって，Nの小さいとき（1,000まで）Shell 分類法は第3.3.3.4節に述べる基数交換分類法よりも効率が良い．Shell 分類法の例は**図3.18**に示されている．

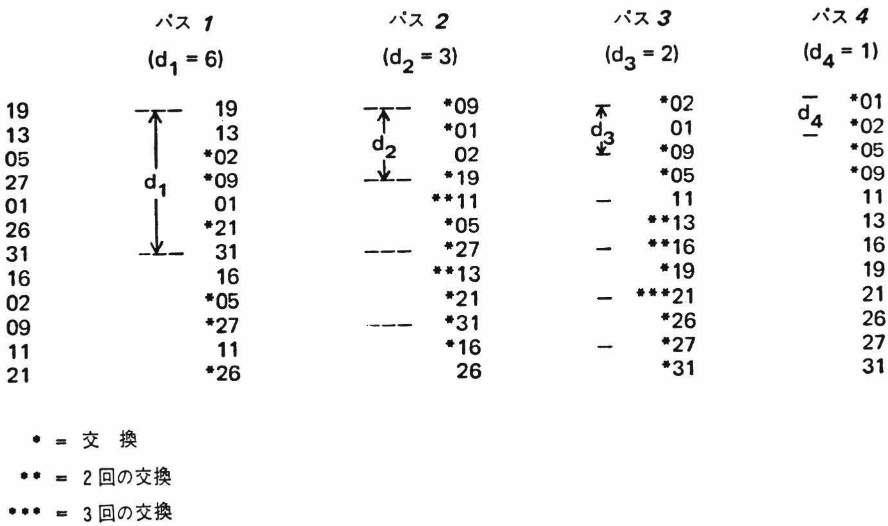


図 3.18 Shell 分類法

3.3.3.3 基数分類法

簡単な分配法の１つは**基数分類法**（radix sort）または**バケット分類法**（bucket sort）と呼ばれるものである．この分類法では見出し語の**最下位**の桁を最初に調べ，その値にしたがって入れ物（バケット）を割当てて行く．すべての項目が分配されたら“入れ物”の中の項目は順番に合併される．同じ処理が見出し語の桁がすべて終了するまで上の桁に向って繰り返される．基数Ｐのとき P 個の入れ物が必要である．

例として**図 3.19** に示された数を基数分類法によって分類してみよう． 読者にはこの分類がどのようにして行なわれるかが直ちにわかるであろう．この方法はカード分類機で行なわれていた方法そのものである．しかし，ディジタル計算機（またはテープを用いて）でこの方法を行なうには大きな欠点がある：

⑴ 分離と合併の２つの別の処理を要する．

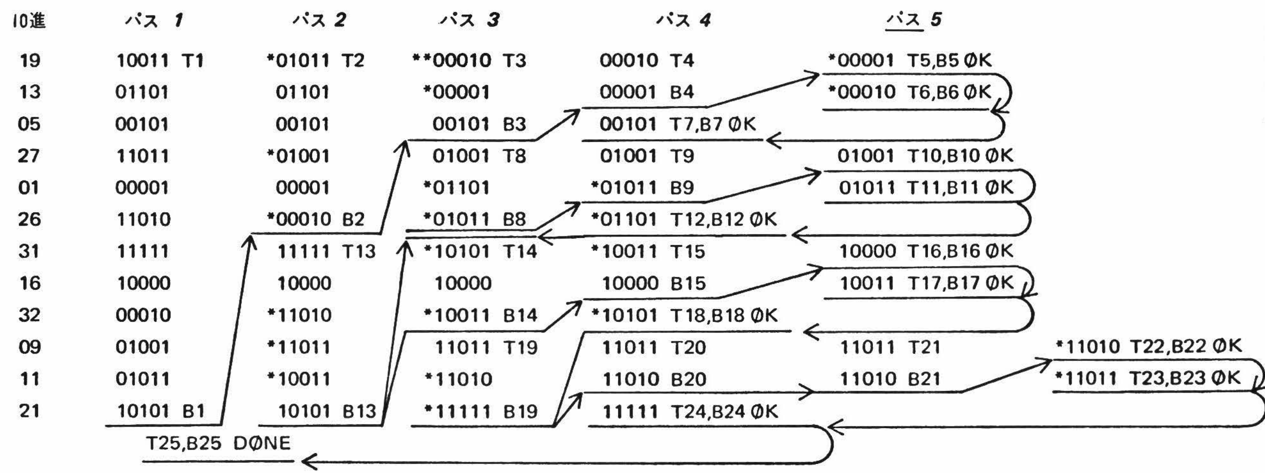
⑵ 入れ物用として余分の記憶装置を要する．

しかし第２の欠点は，最大容量の容れ物をあらかじめ割付ける代りに，論理的な“入れ物”の中でレコードを連結することにより解決できる．

元の表	第1回の分配	併　合	第2回の分配	最終回の併合
19		01		01
13	0)	31	0) 01,02,05,09	02
05	1) 01,31,11,21	11	1) 11,13,16,19	05
27	2) 02	21	2) 21,26,27	09
01	3) 13	02	3) 31	11
26	4)	13	4)	13
31	5) 05	05	5)	16
16	6) 26,16	26	6)	19
02	7) 27	16	7)	21
09	8)	27	8)	26
11	9) 19,09	19	9)	27
21		09		31
↑		↑		
最下位の桁で分離		上位の桁で分離		

図 3.19 基数分類法

3.3.3.4 基数交換分類法



* は交換が行なわれたことを示す

図 3.20 基数交換分類法

ずっと良い分配法は見出し語が２進数で表わされている（または表わされ得る）ときに適用できる**基数交換分類法**である．この分類は，最初のMビットが同じものの群をM+1番目にビットに関して順番に並べることによって行なわれる．与えられたビットに関する群の順番付けは，群の上から探してビットが１のものを求め，下から探してビットが０のものを求め，これらを置換えて続けられる．このアルゴリズムでは，プログラムは多数の群を記憶しておかなければならず，難しいプログラムとなり，余分に長さNの表が必要である．しかし，最適化されたコーディングでは，群は，単に表の最初と見出し語の各ビットに対応する１つの分割点の表とを管理することで記憶することができる.（したがって，32ビット語の場合33の欄が必要である.）**図3.20**に基数交換分類法の例が示されている．この例はやや複雑であり，理解するのが幾分困難である．複雑なことは他の分配分類法にも共通した性質である．

１つの群がただ１つの項目のみを持つとき分類が終了するようにすると，基数分類法の所要時間が $N*\log_P(K)$ であるのでに対して，**基数交換分類法**の所要時間は $N*\log(N)$ である（ただしKは見出し語の最大長で，Pは基数である)．なお，この基数交換分類法は，“入れ物”として余分の領域を必要としない．

3.3.3.5　番地計算分類法

最後の例は**番地計算分類法**である．この方法は十分な記憶容量が得られれば最も速い方法の１つである．分類は，見出し語をそれが“表わす”表の番地に変換することによって行なわれる．例えば，見出し語が４文字長であったとすると，表の適当な番地を計算する１つの方法は，見出し語を表の長さで除し，１つの項の長さを掛け，表の開始番地を加える．表の長さが２の幂乗のとき除算はシフトになる．同じ番地に２つ以上の見出し語が対応しないとすれば，この分類法の所要時間は N*（番地を計算する時間)にすぎない．しかしながら一般にはこのようなことはなく，幾つかの見出し語は同一の番地に対応してしまう．

したがって，１つの項目を計算した番地に入れる前に，まずその場所が既にふさがっているか否かを判定しなければならない．もし使用されていれば，項目はそこにある項目と比較され，新しい項目を入れるべき場所を探すために逐次探索が実行される．幸運な場合には，場所が空いていて正しい順で項目を入

れることができる．場所が空いていないときには，それ以前の項目を移動させて空きを作らねばならない．この種の分類法の実行時間を増大させるのは探索と移動である．

この分類法の所要時間は，収容するデータの数よりも表を大きくすることにより小さくすることができる．表を大きくすることは表の中の空いた領域を大きくし，番地のぶつかりと長い部分の探索や長い部分の移動の確率を小さくする．表の容量を分類すべきデータ数の約2.2倍にすると，この方法の所要時間はNに比例したものとなり，最も速い分類法となる（最後に表の詰め合せを行なわなければならない）．

図3.21に番地計算分類法の例を示す．表の大きさは12であり，最大の見出し語の値は36以下であることがわかっているので，番地変換は見出し語を3で割り整数部を求めることになる（すなわち，$19/3=6+{}^{1}/_{3}$ よって6とする）．“*”は見出し語のぶつかり合いを示しており，矢印は移動が必要であったこととその方向を示している．計算された番地は第2行目に示されている．

データ番号	=	1	2	3	4	5	6	7	8	9	10	11	12
データ	=	19	13	05	27	01	26	31	16	02	09	11	21
計算した番地	=	6	4	1	9	0	8	10	5	0	3	3	7
表	=												
0		—	—	—	—	01	01	01	01	*01	01	01	01
1		—	—	05	05	05	05	05	05	↓02	02	02	02
2		—	—	—	—	—	—	—	—	↓05	*05	*05	05
3		—	—	—	—	—	—	—	—	—	09	09	09
4		—	13	13	13	13	13	13	13	13	13	11	11
5		—	—	—	—	—	—	—	16	16	16	13	13
6		19	19	19	19	19	19	19	19	19	19	16	16
7		—	—	—	—	—	—	—	—	—	—	↓19	*19
8		—	—	—	—	—	26	26	26	26	26	26	21
9		—	—	—	27	27	27	27	27	27	27	27	26
10		—	—	—	—	—	—	31	31	31	31	31	27
11		—	—	—	—	—	—	—	—	—	—	—	↓31

図 3.21　番地計算分類法

3.3.3.6　分類法の比較

これまでに5種の分類法，互換分類法，基数分類法，基数交換分類法，Shell

分類法，および番地計算分類法を論じた．各々の特徴は次に示されている：

分類法	平均時間（近似値）	余分に必要な記憶容量
互換分類法	$A*N^2$	なし
Shell 分類法	$B*N*(\log_2(N))^2$	なし
基数分類法	$C*N*\log_p(K)$	$N*p$
基数交換分類法	$D*N*\log_2(N)$	$k+1$
番地計算分類法	$E*N$	$2.2*N$（近似値）

ただし，Nは表の大きさ，Kは見出し語の最大長（IBM 360では普通32），Pは基数分類法のときの基数，A，B，C，D，Eは比例定数である．

比較分類法はN^2ないし $N*\log(N)$ に比例した時間がかかる．この時間はデータの値の分布のし方に影響され，データ内の順番付けをそのまま利用でき，余分の記憶容量を必要としない．

分配分類法は，判定しなければならないN個の数があり，1つの数は$\log_p(K)$桁の数字から構成されているので，ほぼ $N*\log(N)$ に比例した時間がかかる（ここではPは基数，Kは見出し語の最大長）．しかし，分配分類法は値の分布のし方に影響され，データ内の順番付けはほとんど利用できず，場合によってはかなり余分の記憶容量を必要とする．

番地計算分類法は，表の欄に最初に入る要素に対しては番地のぶつかり合いが生じないのできわめて速い．しかし，表が埋まって行くにつれて，新しい欄を追加する時間は指数関数的に増大する．

要約すると，**互換分類法**は最も単純であり，速度が問題とならない場合に使用されるべきである．**基数分類法**は時間の効率は良いが，法外に記憶容量を必要とするので計算機ではめったに使用されない．カード分類の場合には領域の問題はないのでこれは良い方法である．**基数交換分類法**は非常に速く余分の記憶容量もほとんど必要でない（IBM 360　では大体32語）が，プログラミングとデバッグが難しい．**番地計算分類法**は，効率を良くするには最も大きな領域が必要であるが，このような領域が得られれば他の方法より高速である．

3.3.4　ハッシュ法による登録と探索

2分探索法は高速であるが**順番に並べられ，詰め合された**表，すなわち，隣り合った項目が見出し語の順に並んだ表だけにしか適用できない．したがって，この種の探索手続きはデータを順番に並べ合せる分類法と共に使用しなければならない．

しかし実際には，探索を能率良く行なうために表が順番に並べられ，詰め合されている必要はない．直ちにわかるように，表にすき間があるとき，すなわち，表の領域が格納される項目の個数より多いときには，詰め合されず，順番に並べられていない表を用いて比較的良好な探索を行なうことができる．

前に番地計算分類法は，表をすき間があるとき，良い結果を与えることを見た．しかし，順番に要素を入れなければならないことは処理速度を低下させた．要素をランダム（または擬似的にランダム）に入れることにより能率を大幅に改善することができる．ランダムな欄の番号Kは，番地計算分類法に使用されたのと同じような方法で，見出し語から生成される．K番目の欄が空いていれば，新しい要素はそこに入れる．もし空いていなければ，挿入のために他の欄を探さなければならない．

まず問題となるのは見出し語から乱数を作ることである．もちろん，この場合には同じ見出し語が今日と明日とで違った場所を指定するような乱数を望んでいるのではない．1つの見出し語に対しては同じ表の位置を示す擬似乱数を生成する手続きが必要なのである．4つの EBCDIC 文字の見出し語に対して良好な結果を与える方法の1つは，単に表の長さNで除し，剰余を求めることである．この方法はNと見出し語の大きさ（IBM 360 では32ビット）が公約数を持たない限り良好に働く．M個の見出し語の群に対して，剰余は0……（N—1）にかなりの程度均等に分布するであろう．他の方法は見出し語を2進小数と考え，これに次の演算を行なう．

```
L     1,SYMBOL
M     0,RHO
```

結果はレジスタ0と1の中に置かれた64ビットの積である．RHO を注意深く選ぶと下位の31ビットの値は0と1の間にほぼ均等に分布する．次にNを乗ずると0……（N—1）の範囲で均等に分布する数が生成される．この方法は**乗積法**（power residue）と呼ばれる．この方法の利点は，表の欄が既に使用され

ているとき，上位の31ビットの結果が**再び**一様分布の数を（RHO を再び乗じて）生成するために使用できることである．

第2の問題点は，既に使用されている場所にぶつかったとき，次にとる手続きについてである．この問題を解決するのに幾つかの方法があるが，ここではそのうちの3つについて論じる：

1. 単純乱数法　一連の乱数を乗積法のようにして見出し語から生成し，これらによって1とNの間の数を求めて表を調べる．空いた場所が見付かれば終了する．注意すべきことは生成される乱数は互いに独立であり，同じ位置を2回調べることもあり得ることである．

2. 重複を省く乱数法　この方法は，同じ位置を2回調べないという点以外，上の方法と同様である．この方法は判定の経費が高いときのみ，例えばテープやドラムのファイルに対して，有効である．

3. 開放番地法　最初位置Kが与えられたとし，その位置がふさがっていたとすると次の場所 K+1 を調べる．これを空いた場所が見付かるまで繰り返す．探索していて表の終りになれば最初から始めることにする（すなわち，表は円筒になっていると考える）．

これら3つの内**開放番地法**が最も簡単である．次の例はこの方法を説明している：

17の欄を持つ表（N=17）に次の12の数を格納することを考える．19, 13, 05, 27, 01, 26, 31, 16, 02, 09, 11, 21．これらの項目は，17で除した剰余で定まる位置に記入される．その位置がふさがっていれば，次の位置が調べられる……**図3.22**は12の項目に対してどのように欄が発展するかを示している．02, 09 および11の項目にぶつかり合いの処理が見られる．“判定回数”の列は表の中の対応した項目を探すために必要な判定回数を示す．したがって，項目09を探すために3回，項目26では1回の判定が必要である．“空欄判定回数”の列は表の中の空いている欄を探すのに必要な判定回数である．したがって，数54のときには最初の位置は3で，空いている位置を求めるのに4回の判定が必要であることがわかる（この場合位置6）．次の値が得られる：

表の長さ	N=17
格納項目	M=12

位置	項目	判定回数	空欄判定回数
0			1
1	01	1	6
2	19, 02*	1	5
3	02	2	4
4	21	1	3
5	05	1	2
6			1
7			1
8			1
9	26, 09*	1	7
10	27, 09*	1	6
11	09, 11*	3	5
12	11	2	4
13	13	1	3
14	31	1	2
15			1
16	16	1	1
		16	54

図 3.22 開放番地法

密度	$\rho=12/17=0.705$
格納するために要した判定回数	$T_s=16$
平均判定回数	$T_p=16/12=1.33$
平均空欄判定回数	$T_n=54/16=3.37$

詰め合された表に対する基数変換分類法および2分探索法の時間を比較すると，

格納および分類の判定回数	$T_s=M+M*\log_2(M)=55$
平均判定回数	$T_p=\log_2(M)=3.58$
平均空欄判定回数	$T_n=\log_2(M)=3.58$

したがって，開放番地法はかなり速いという利点を有するが，表の容量が50%近くも余計に必要である．さらに，いったん表を割付けると表を縮めることはできず，割付けられた領域を幾つかの表で共有することもできない．最後に1つきわめて大きな欠点がある．これは表から項目を**削除**することが困難であることである．番地付けの鎖を破壊するので，その場所を単純に抹消することはできない．

乱数法に対して判定時間の期待値を考察してみよう．計算が最も単純な方法

は**単純乱数法**である．K－1項目が既に記入されているN欄の表を考える．密度を $\rho=(K-1)/N$ と定義すると，

K番目の場所に格納するための判定回数 $L_K=\dfrac{1}{1-\rho}$

探索のための判定回数 $T_P=\dfrac{1}{\rho}\log_e\dfrac{1}{1-\rho}$

これらの値は，探索時間と表の密度の関係をよく表わしていてきわめて興味深い．

開放番地法に対してこの値はかなり異なっている．表が密になるにつれて長い列となる確率は増し，判定回数が増加する．判定の回数は約

$$T_P(\rho)=1+\frac{\rho}{2}\cdot\frac{1}{1-\rho}$$

項目が表の中にないことを決定するための判定回数は約

$$T_N(\rho)=\frac{1}{1-\rho}$$

密度 $\rho=12/17$ のときこれらの値は，

$$T_P=1+\frac{6}{5}=2.2$$

$$T_N=\frac{17}{5}=3.4$$

これらの値は前の例に比べて良好である（詳細な数値に関しては第10章参考文献を参照）．

擬似乱数の生成

擬似乱数の生成には多くのアルゴリズムがあるが，乗積法は最も一般に使用されている．これについては Hamming（第32章第5節）または Kunth の本を参照してほしい（第10章参考文献を参照）．

3.4 まとめ

本章では2パス・アセンブラの設計を示した．パス1では記号を定義し，パス2では目的デックを生成する．アセンブラや他のソフトウェアを設計するのに際しては次の6つの基本段階を追って行かなければならないことを強調した：

1. 問題を設定する.
2. データ構造を規定する.
3. データ構造の書式を定義する.
4. アルゴリズムを規定する.
5. 機能単位を探す.
6. 1から5を各モジュールについて繰り返す.

アセンブラの特に重要な構成要素は各種のデータ・ベース(例えば,記号表,機械命令表)である.表を構成し,探索し,そして分類する技法を述べた.

3.5 あとがき

読者の中には"プログラミングは科学でなく芸術である"という言葉を聞いた人があるかも知れない.もちろんプログラミングの方法には,ちょうど文を書いたり話したりするのに各人各様の違いが見られるように,各種のスタイルがある.しかし,アセンブラの設計で示したように,基本的な設計の過程は,通常,容易に理解できるものであり,容易に理解できるものでなくてはならない.

したがって,ソフトウェア設計者の行動を何か神秘的な"魔術"のように考えるのは誤っている.例えば,本章では紙面の多くを探索および分類の技法の説明に費し,印刷の書式やアセンブラの他のモジュールの詳細を省略した.これらは適当に決定してよいと思えるだろう.探索や分類の技法に注目したのは効率に関して重要で,**あい路**となりかねないという点に原因している.すべてのアセンブラの機能モジュールは同程度にプログラムが複雑である.しかし,印刷書式モジュールはカード1枚につき1回使用されるのに対して,例えば,記号表探索モジュールは各原始カードについて**何百回何千回**と実行されるループを持っている.ソフトウェア設計者は有限の時間しか持たないので,効率に最も影響を与える小数のモジュールに重点を置くことが絶対必要である.

簡単な例を用いてこの点を説明する.典型的な命令実行時間が12マイクロ秒(すなわち,毎秒平均83,000命令実行)の IBM 360 モデル40などの中速の計算機でアセンブラを実行したとする.かなり大きく,カードが5,000枚で約2,000の記号を持つアセンブラ原始プログラムをアセンブルしたいとする.各

原始カードは演算数部に平均1個の記号参照を行なっているとき，記号表の探索に費される時間を計算してみよう．

逐次探索法が**図 3.12** のプログラムで実行されたとすると， 探索の各ループで5つの命令が実行される．したがって，各繰り返しについて $5 \times 12 = 60$ マイクロ秒かかる．繰り返し回数は大体表の半分であるから $1000 = (1/2)*2000$．そして 5,000 枚の原始カードの各カードに1回の探索があるので，探索の総時間は，

$$
\begin{aligned}
\text{総探索時間} &= \text{探索回数} \times \text{ループの繰り返し回数} \times 1\ \text{回の繰り返し時間} \\
&= (5\times10^3)\times(10^3)\times(60\times10^{-6}) \\
&= 300\text{秒} \\
&= 5\text{分}
\end{aligned}
$$

一方， 2分探索法を使用すると， 1回の探索に必要な繰り返し回数はただの $\log_2(2000)-1 \approx 10$ であり，**図 3.15** のようにプログラムすれば， 各繰り返しの所要時間は 100 マイクロ秒である．

$$
\begin{aligned}
\text{総探索時間} &= (5\times10^3)\times(10)\times(100\times10^{-6}) \\
&= 5\ \text{秒}
\end{aligned}
$$

このように2分探索法はアセンブル時間を4分55秒短縮する（別のいい方をすれば，逐次探索法は 6,000% の時間がかかる）．さらに，アセンブラは日に何回となく使用されるので，アセンブル時間の節約は計算機システムの生産性を大きく改善する．

以上はあい路解析の非常に簡単な例であったが，探索および分類の技法の重要性を指摘している．経験的には大半のプログラムの実行時間の 80% は 20% 以下のモジュールによって費されていることがわかっている．したがって，あい路の解析はソフトウェア設計者の最も重要な技巧となっている．

演 習 問 題[1]

1. アセンブラ語のどのような特徴のためにアセンブラを2パスにする必要が

---

1 ＊は IBM 360マニュアルの使用を要する問題であることを示す．

あるのか.

2. 3パス・アセンブラになるとどのような特徴が加わるか.

3. 次の機能を果すアセンブラ・プログラムのパスの番号を示せ：

	機 能	パス
a.	ベース・レジスタ表を保存する.	1 2
b.	ロケーション・カウンタを更新する.	1 2
c.	記号の定義を格納する.	1 2
d.	START 擬似命令を処理する.	1 2
e.	中間テープを書出す.	1 2
f.	2進形式のカードを穿孔する.	1 2
g.	END 擬似命令を処理する.	1 2
h.	L命令コードを処理する.	1 2
i.	DS 擬似命令を処理する.	1 2
j.	DC 擬似命令を処理する.	1 2
k.	EQU 擬似命令の演算子部に入っている記号を探す.	1 2

4. TTS（Tepid Terminal System あまりやる気のない TSS 会社の意味）は，1パス・アセンブラを端末サービスのために設計するという契約を読者との間に交した. これは IBM 360 に組込まれる予定である. 読者はアセンブラの仕様を作成しなければならない. 次の機能は無条件に組込めるか，不可能か，またはどのような制限条件のもとで組込めるかを示せ.

   a. USING, DROP
   b. START
   c. EQU
   d. DC, DS
   e. 通常の記号名の命令コード
   f. EXTRN
   g. 命令に付けるラベル
   h. "B LABEL" という命令. ただし LABEL はプログラムの後の部分で定義される.

5. *a. USING 命令を用いずにアセンブラ語のプログラムを書けるか．できるとしたらどのようにするのか．制限事項は何か．

b. 次は何のために使用される命令か．

DS 0D

6. 本章で示された LTORG 擬似命令をアセンブラに追加したい．プログラム中の LTORG は，前の LTORG 以後（または前に LTORG がなければ START 以後）現われたすべてのリテラルを，現在の位置に最も近い2語境界から開始して生成する．パス1とパス2にどのような変更が必要か．LTORG の処理はどうして1パスで行なえないのか．

7. a. DC 擬似命令の長さや繰り返しの係数に現われる記号はなぜ自己定義の，または，それ以前で定義されている記号でなければならないのか．

b. EQU 擬似命令の演算数部の値はなぜパス1で計算しなければならないのか．

c. DC 擬似命令の演算数部で定義されている定数はパス2までは生成されず，プログラムに挿入されないか．なぜパス1でこれを行なえないのか．

8. IBM 360 のアセンブラ語の“拡張記号”によって BC（条件分岐）命令の特殊な場合には省略形が使用できる．例えば，

BZ × ≡ BC 8, ×

BM × ≡ BC 4, ×

ここで BZ は零で分岐，BM は負で分岐を意味する．

a. どのようにしたら本章で論じたアセンブラに拡張記号を追加できるか簡単に述べよ．余分のパスを追加してはならない．

b. この変更はパス1のみか，パス2のみか，あるいは両方か．その理由は何か．

9. 次のプログラムに対して

1） パス1終了時の記号表を示せ．

2） パス1終了時のリテラル表を示せ．

3） パス2でのベース表の変化を示せ．

4） パス2で生成される機械コードを示せ．

a.

```
         SIMPLE   START
                  BALR     15,0
                  USING    *,15
         LOOP     L        R1,TWO
                  A        R1,TWO
                  ST       R1,FOUR
                  CLI      FOUR+3,4
                  BNE      LOOP
                  BR       14
         R1       EQU      1
         TWO      DC       F'2'
         FOUR     DS       F
                  END
```

*b.

```
         SAE      START    484
         ARCHON   EQU      1
         DEPUTY   EQU      2
         TREAS    EQU      3
                  BALR     2,0
                  USING    *+ARCHON-DEPUTY-TREAS,TREAS
                  LM       1,6,POINT
                  USING    BETA,ARCHON,DEPUTY
                  EXTRN    BACK
                  ENTRY    BEACON
         BEACON   CR       DEPUTY,ARCHON
                  BNH      POINT+SAE-BEACON
                  LA       7,=A(BACK)
                  CLI      HOLE,X'90'
                  BR       6
                  DC       H'64',X'40',B'1000000',C' '
                  DROP     DEPUTY
                  DC       15X'0',6F''80'
         BETA     MVC      POINT+20(4),=H'43'
                  MVC      POINT+16,=H'43'
         STOMP    EQU      5
                  DROP     ARCHON
                  L        9,=A(BEACON)
                  NR       9,TREAS
                  ST       9,=F'482'
                  B        POINT+24
                  LTORG
         HOLE     DS       0D
         POINT    DC       CL4'WIN'
                  DC       (STOMP)A(BETA),V(FOOTBL)
                  CLC      POINT-4,=F'482'
                  LA       1,1
                  CVB      2,10(DEPUTY*TREAS)
                  STC      3,POINT(TREAS*TREAS)
                  BR       14
                  END
```

10. 本章で論じたアセンブラに組込みたい IBM 360 アセンブラ語の特徴をこの問題で考えてみる．

複数個の同一書式の欄を持つ表を持っているとする．この表の中の欄の番地指定を，各欄の先頭を指すベース・レジスタとそのレジスタからその欄の各部分への変位とを用いて行ないたい．この場合にはプログラマが変位を記憶しておかなければならないという欠点がある．ここでほしいのは，各部分を記号で参照できることとアセンブラがベースと変位を他の記号と同じようにして計算してくれることである．

DSECT 擬似命令を用いてこの変位を定義することにしよう．この命令は初期値 0 の**新しい**ロケーション・カウンタを作り， DSECT に続く記号に対して，アセンブラがちょうど DSECT の前の記号には**古い**ロケーション・カウンタに相対的に番地を割付けたと同様にして，新しいロケーション・カウンタを基にした相対番地を割付ける．この問題で DSECT はプログラムの終りに置かれ，他の DSECT または END 命令は一番新しく作られたロケーション・カウンタの使用を終了させるものとする．

DSECT の内部で定義される記号を参照する命令を生成するには，USING 擬似命令を用いて，選択すべきベース・レジスタを指定する． USING の番地部には DSECT 命令のラベルが入り，ベース・レジスタはこれまでと同様に指定される．

例：

ロケーション・カウンタ	生成されたコード			
0		SUB	START	0
0			USING	SUB,15
0			USING	TABLE,3
0	LH 4, 10(0,15)		LH	4,FIFTY
4	LH 5, 4(0,3)		LH	5,B
8	BCR 15,14		BCR	15,14
10	H'50'	FIFTY	DC	H'50'
0		TABLE	DSECT	
0		A	DS	F
4		B	DS	H
6		C	DS	H
			END	

DSECT に続く文では新しいロケーション・カウンタが更新され，それに

より記号が定義されるが記憶場所は使用しないこと，また，FIFTY は 10（0,15）で置換えられ，Bは 4（0,3）で置換えられていることに注意してほしい．

DSECT の概念を処理するのにアセンブラの 3つの部分を変更する必要がある．

1）DSECT 命令を識別し処理を行なうこと．
　　DSECT の定義域内で定義されるラベルを処理すること．

2）DSECT を参照する USING 命令を処理すること．

3）命令の演算数として使用されている記号を処理して，ベースと変位により置換えること．

これらの各部分について次の問に答えよ．

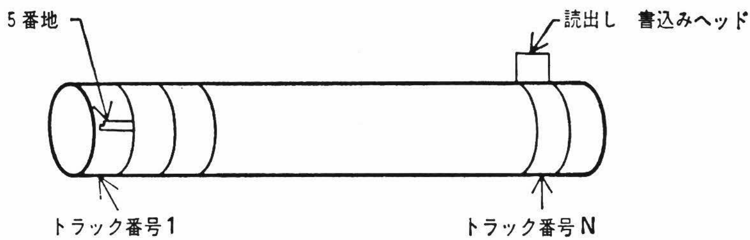
a. アセンブラのどのパスが変更されるか．その理由は何か．

b. 追加したり変更したりする必要のある論理は何か．

c. 現在のデータ・ベースにどのような修飾が必要か．その理由は何か．

11. 古い計算機会社は仕事がなくなり唯一の製品を残した．この計算機はコア記憶装置がなく，ドラムが付いている．2つの作業レジスタ AC と MQ（インデックス・レジスタはない）がある．AC は加算と減算に使用され，AC/MQ は対になって乗算および除算に使用される．機械は26 ビット語で動作する．命令形式は1つで次に示すとおりである．

命　令	トラック	番　地	トラック	番　地
0　　5	6　　10	11　　15	16　　20	21　　25



機械はビット 0～6 を命令コード，ビット 7～16 を演算数の記憶場所，ビット 17～25 を次の命令の記憶場所として解釈する．

この計算機のアセンブラ語プログラムの例を示す：

```
          LOAD    A,LOC1     アキュムレータにAをロードする
LOC1      ADD     C,LOC3     Cをアキュムレータに加える
A         DC      F'5'
LOC3      STO     D,LOC10    アキュムレータの内容をDに格納する
C         DC      F'4'
LOC10     STOP
D         DS      F
          END
```

変数部の第1の部分は演算数の番地で，第2の部分は次の命令の番地である．ドラムはつねに回転している．実行時間を最小化するためには命令をトラックの上で4番地離れた場所に置けばよいことがわかっている．このようにするとドラムの“読出し”ヘッドが次の命令の所へくる前に命令の実行が完了する．

次の点について，本章で論じたアセンブラにどのような変更が必要かを簡単に述べよ．

a. ロケーション・カウンタの書式．

b. ロケーション・カウンタの更新と維持．

c. 実行時間を最適化するために，次の各々について記憶領域の割付けをどのようにすればよいか．アセンブラに課す制限とプログラマに課す制限とを区別せよ．

　1）命令

　2）DC および DS 擬似命令

　3）リテラル

d. このように最適化したときロケーション・カウンタはどのように維持されるか．

e. 本章で述べたアセンブラのパス1とパス2をどのように変更すればよいか．

f. 2パス・アセンブラは必要か．その理由は何か．

12. a. 次の術語を1～2文で定義せよ．

　1）探索法

　2）分類法

3) ハッシュ法

b. 探索と分類の相違点は何か.

13. 次の分類法による各パスの結果を示せ.

1) 互換法

2) Shell 法

3) 基数法

a.	b.
81	424
52	887
57	807
22	709
95	882
04	616
83	573
96	413
42	679
32	180
48	975
78	264
82	
65	
16	
66	
14	
77	
87	
67	

14. 次の表はアルファベットの順に並べられている. 2分法が使用されるとして表の中で名前を探せ. 各欄を探すために必要な判定回数を示せ. 例えば, DONOVAN を探すためには3回の判定が必要である (8=GOODMAN, 4=BRUNK, 6=DONOVAN)

	項目	所要判定回数
1)	ASHTON	
2)	BERMAN	
3)	BERNSTEIN	
4)	BRUNK	
5)	DAVIS	
6)	DONOVAN	3
7)	FREYBERG	
8)	GOODMAN	
9)	KOHN	
10)	MADNICK	
11)	MICHELS	
12)	NANGLE	
13)	STEEL	
14)	TURNER	
15)	ZILLES	

15. 次の表に**図 3.19** と同様に基数分類法を用いたとき，各パスの結果を示せ．

a. 00100
10001
01011
00001
00010
00101
00000
01001
10101
10010
01111
11011

b. 100011
011000
000111
010100
010110
100010
111100

000100
100001
101100
110111
110011
110000
010000
000110
111001

16. 次の表は幾つかの分類法と各種の性質を示している．ここでいう互換分類法とはあるパスで置換えがなされなかったとき停止するモデルをいう．分類法が指定の属性を持つとき対応する欄に✓を付けよ．

分類法 / 属性	Shell 法	基数法	互換法	基数交換表
各判定ごとに見出し語の全てのビット（または文字）を検査する				
余分に大きな記憶域を必要とする				
すでに表が分類されているときただ1回のパスで分類できる				
分配分類法の1つである				
分類のためには項目が2進数で表現されていなければならない				

17. アセンブラのパス1においては，パス1が終了する前にでも，記号表から記号を参照することができれば都合が良い．例えば，擬似命令

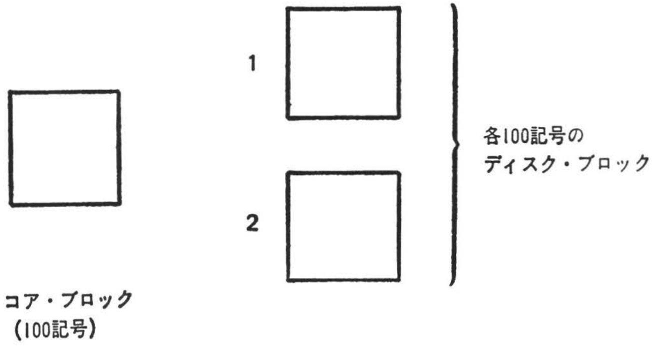
A　　EQU　　B

ではAに対する値を求めるためにBの値を求めることが必要である．この要求を満すために記号表に対するデータ格納と参照の最良の方法を述べてみよ．その理由も説明せよ．

18. 読者には小さな計算機会社の主任プログラマとして，会社の最新式の計算

機に対するアセンブラを作る仕事が割当てられたとする．アセンブルの間に記号表は任意に長くなり得る．読者はこの表をどのようにして作り，どのようにして表の欄を参照すればよいかに悩んでいる．

コアには100個の記号を納める領域しかないが，容量が100のブロックで記号を格納できる2次記憶が付いている．



1）　ディスクのブロックを読んだり書いたりするのに100ミリ秒（10^{-1}秒）かかる．

2）　平均の命令実行時間は10マイクロ秒（10^{-5}秒）である．

a.　記号表の欄の個数が100以下と仮定して表を並べるとすれば，分配法と比較法のいずれの分類法を使用するのがよいか．1000以上ならどうか．その理由は何か．

b.　次の表の上に示してある4つの場合の各々に対して，記号表に欄を作ったり読出したりするのにオーバーヘッドが最小であると考えられる登録と参照の技法を✓で示せ．ただし，各場合とも必らずしも1つだけが正しい答えとは限らない．各場合について，それを選んだ基準を2つの文で説明せよ．（ヒント：100欄以上の場合には表のブロックを入換えるのにかなりの時間がかかることに注意.）

19. 下に次の名前，各々はIBM 360型計算機で2語（8バイト),の表を分類するためのハッシュ・コードを計算する流れ図を示す．

DONOVAN*b*

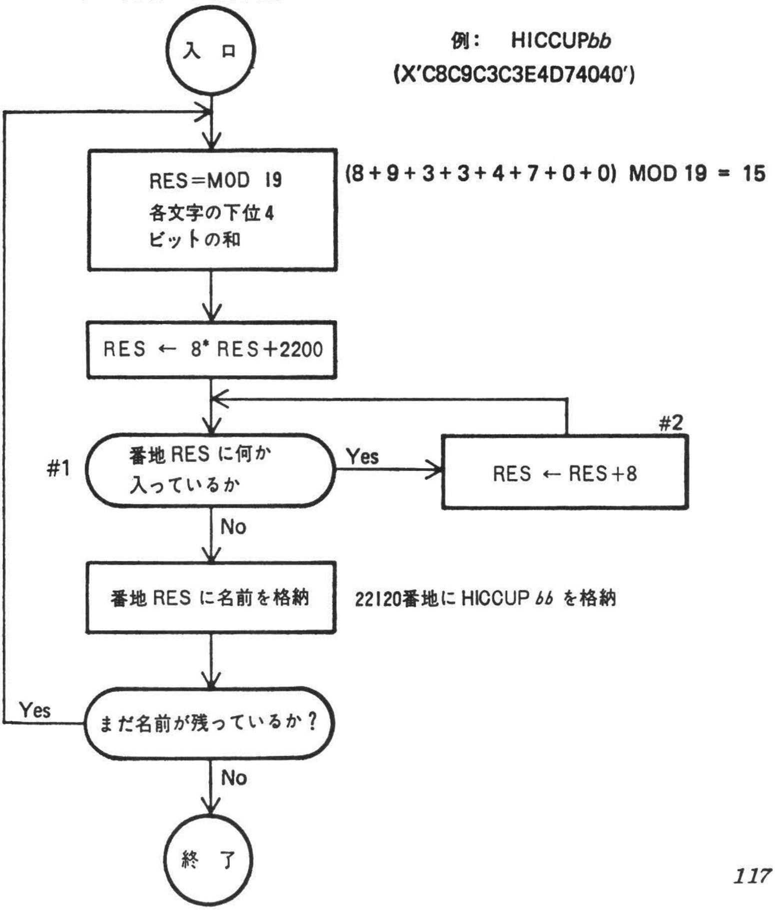
MULHERN*b*

		場合1	場合2	場合3	場合4
		表の大きさ <100 表はそんなに頻繁に参照されない	表の大きさ <100 表は頻繁に参照される	表の大きさ >1000 表はそんなに頻繁に参照されない	表の大きさ >1000 表は頻繁に参照される
表の参照法	逐次探索法				
表の参照法	2分探索法				
表の参照法	ハッシュ探索法				
表の参照法	2分法付きハッシュ探索法				
表の管理法	順次記入				
表の管理法	順次記入後互換分類を行なう				
表の管理法	順次記入後基数分類を行なう				
表の管理法	順次記入後 Shell 分類を行なう				
表の管理法	ハッシュ法で記入				
表の管理法	ハッシュ法で記入、欄がぶつかれば分類を行なう				

FREYBERG
MITCH*bbb*
HARRIS*bb*
MARY*bbbb*
MADNICK*b*
NANGLE*bb*

a. 分類の後での番地 22000 から 22160 の内容を記せ.

b. 2つ以上の名前が同一の場所に割当てられるという問題は，ブロック 1と2で次の場所を判定して処理される．この問題の他の処理方法をあげ利点と欠点を論ぜよ.



20. a. 次の姓名の表を，姓だけについて Shell 法を用いて辞引式の順番に分類せよ．同じ姓があれば名前に拘らず同一であるとして処理せよ．並べられた表の 1 例は：

1	DONOVAN	JOHN
2	FAY	REGAN
3	FÄY	ELIZABETH
.		
.		
.		
N	QUIMBY	DAVE

←TABLE (3,2) = ELIZABETH

分類されるべき表：

FREYBERG, DUTCH
ZILLES, STEVEN
KOHN, NORMAN
SINNOTT, MARY
BRUNK, GLEN
DONOVAN, MAUREEN
DONOVAN, JAMES
DONOVAN, CAROLYN
DONOVAN, REBECCA
DONOVAN, JOHN
DONOVAN, MARILYN
MADNICK, STUART
NANGLE, ELLEN
GOODMAN, LEONARD
DODSON, ORVILLE
ASHTON, BENJAMIN
JOHNSON, JERRY
HAMMER, MICHAEL
JACK, MARTIN
BERMAN, HARRIS
RAMCHANDANI, CHANDER

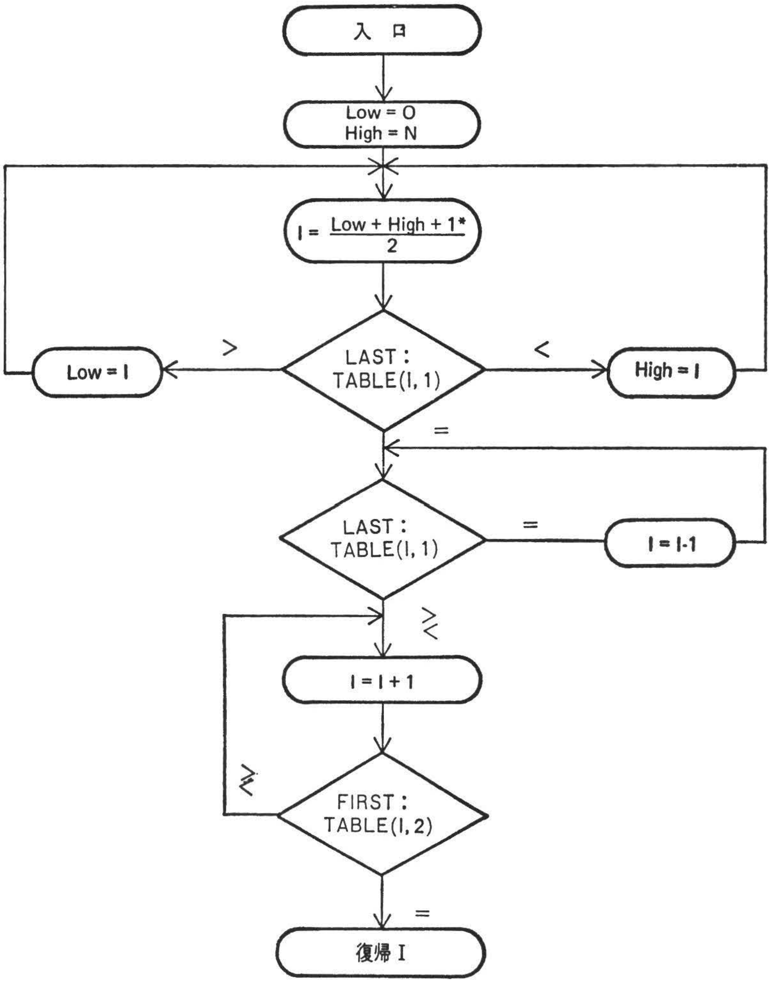
次は欄を探す流れ図である．問 b，c に対して問 a で並べた表を使用せよ．

b. N=22，FIRST=MAUREEN および LAST=DONOVAN に対してこのアルゴリズムは何を返却するか．

c. このアルゴリズムはどの探索法を使用しているか．この探索アルゴリズムの持つ属性について述べよ（時間，所要記憶容量など）．

d. この探索アルゴリズムの利点は何か．

e. この探索アルゴリズムの欠点は何か．


入 口
Low = 0
High = N
I = Low + High + 1* / 2
LAST : TABLE(I, 1)
>
Low = I
<
High = I
=
LAST : TABLE(I, 1)
=
I = I-1
≷
I = I + 1
FIRST : TABLE(I, 2)
≷
=
復帰 I

* 必要なら整数に切捨て

第 4 章

マクロ言語とマクロ・プロセッサ

アセンブラ語のプログラマは，プログラムを作成するとき，しばしば同じ命令の組を何回も繰り返さなければならないことがある．この命令の組は，例えばレジスタを保存したり入換えたり，サブルーチンの結合を行なったり，一連の算術演算を行なったりするコードから成っている．この場合プログラマにとってはマクロ命令の機能が役に立つ．**マクロ命令**(しばしば**マクロ**と呼ばれる)は一群の命令を1行で書くことができるような省略形である．マクロ命令を用いるためにプログラマは1つの“命令”が一群の命令を代表すると定義する．この1行のマクロ命令がプログラムの中で現われる度に，マクロ・プロセッサは定義ブロックを代入する．

適切なマクロ命令を定義することによって，アセンブラ語のプログラマは費用をかけずに，プログラムの構造を具合よく制御できる自分専用の高級な設備を仕立てることができる．プログラマはアセンブラ語によるプログラミングの基本的な利点を失うことなく，高水準言語でコーディングする簡潔さと容易さを手に入れることができる．完全なマクロ処理によってデバッグおよびプログラムの修飾が簡単になり，標準化が容易になる．多くの計算機会社はマクロ機能を用いる**システム編集**によって“特別仕立て”のオペレーティング・システム作成を自動化して行なっている．

マクロ命令は，通常，基本アセンブラ語の拡張と考えられ，マクロ・プロセッサは基本アセンブラの拡張と考えられる．しかしながら，プログラム言語としてのマクロ命令はアセンブラ語やコンパイラ言語とは大きな違いがある．類似点はある種の高水準言語やテキスト編集システムに見られる．

本章ではアセンブラのマクロ機能について論じ，プログラミングへのマクロ

命令の使用法とアセンブラへのマクロのインプリメンテーションを取り扱う．たいていのマクロ・プロセッサが持つ特徴を紹介しそれを展開して行く．

4.1 マクロ命令

最も単純な形のマクロは一連の操作に対する省略形である．次のプログラムを考えてみよう：

例 1：

```
        ⋮
A          1,DATA    DATAの内容をレジスタ1に加える
A          2,DATA    DATAの内容をレジスタ2に加える
A          3,DATA    DATAの内容をレジスタ3に加える
        ⋮
A          1,DATA    DATAの内容をレジスタ1に加える
A          2,DATA    DATAの内容をレジスタ2に加える
A          3,DATA    DATAの内容をレジスタ3に加える
        ⋮
```

上のプログラムで

```
A       1,DATA
A       2,DATA
A       3,DATA
```

は2回現われている．マクロ機能によって，この命令群に名前を付け，命令群の代りにその名前を使うことができる．上のブロックをマクロとして定義し，後でこの定義を参照できるようにするマクロ言語を作ることができる．ここではIBM 360型のマクロ言語を用いることにする．

マクロ・プロセッサはそれ自身の言語を持つ独立の言語プロセッサである．

マクロ命令定義によって1つのブロックに名前を付けるが，これは次のようにして行なわれる．

```
定義の開始──────────────→   MACRO
マクロ名────────────────→   〔      〕
                             ───────
省略すべき本体          {    ───────
                             ───────
定義の終了──────────────→   MEND
```

MACRO 擬似命令は定義の第1行であり，その次の行が**マクロ命令の名前**であることを示している．名前の行に続いて，省略される一連の命令，すなわち“マクロ命令”を構成する命令，が置かれる．定義はMEND（“マクロ終了”）擬似命令の行で終る．

いったんマクロが定義されると，アセンブラ・プログラムの中でマクロ名を命令記号として使用することは，対応した命令の列を使用することと等価になる．繰り返される列に‘INCR’という名前を付けておくと，前の例は次のように書直すことができる．

```
        原始プログラム              |     展開された原始プログラム
                                    |
        MACRO                       |
        INCR                        |
        A         1,DATA            |
        A         2,DATA            |
        A         3,DATA            |
        MEND                        |
         .                          |
         .                          |            .
         .                          |            .
        INCR                        |   {  A            1,DATA
         .                          |   {  A            2,DATA
         .                          |   {  A            3,DATA
         .                          |            .
        INCR                        |   {  A            1,DATA
         .                          |   {  A            2,DATA
         .                          |   {  A            3,DATA
         .                          |            .
DATA    DC        F'5'              |  DATA    DC       F'5'
         .                          |
         .                          |
         .                          |
```

この場合マクロ・プロセッサは各マクロ呼出しを

```
A     1,DATA
A     2,DATA
A     3,DATA
```

によって置換える．

この置換えの処理をマクロの**展開**という．マクロ定義そのものは展開された原始コードの中に現われないことに注意してほしい．定義はマクロ・プロセッサによって保存されている．展開すべき命令記号として，原始プログラムの中にマクロ名を使用することを**マクロ呼出し**という．

4.2　マクロの機能

4.2.1　マクロ命令の引数

これまでマクロは，マクロ呼出しの所へ命令のブロックを挿入できるものとして説明した．与えられたマクロに対するすべてのマクロ呼出しは同一のブロックによって置換えられた．このマクロ機能は柔軟性に欠ける．マクロ呼出しに応じてそれを置換えるコードを修飾する方法はまだ与えられていない．そこでマクロ呼出しに引数またはパラメータを入れることができるように拡張を行なう．このため，マクロの定義には引数に対応した**マクロの仮引数**が示される．次のプログラムを考えてみよう．

例　2：

```
                 ⋮
        A              1, DATA 1
        A              2, DATA 1
        A              3, DATA 1
                 ⋮
        A              1, DATA 2
        A              2, DATA 2
        A              3, DATA 2
                 ⋮
DATA 1  DC             F'5'
DATA 2  DC             F'10'
```

この場合，各命令列は非常によく似ているが同一ではない．最初の一続きの命令は，DATA1 を演算数として使用し，処理を行なう．第2は DATA2 を

使用している．これらはパラメータを変えて同じ処理を実行していると考えることができる．このようなパラメータは**マクロ命令の引数**または**仮引数**と呼ばれる．引数はマクロ名の行で定義され，第1文字に“&”記号を置いて（アセンブラ語の記号でなくマクロ言語の記号として）識別される．前のプログラムは次のように書ける：

原始プログラム				展開された原始プログラム		
	MACRO		マクロINCRは1つの引数を持つ			
	INCR	&ARG				
	A	1,&ARG				
	A	2,&ARG				
	A	3,&ARG				
	MEND					
	⋮				⋮	
	INCR	DATA1	DATA1を演算数として使用する		A	1,DATA1
					A	2,DATA1
					A	3,DATA1
	⋮				⋮	
	INCR	DATA2	DATA2を演算数として使用する		A	1,DATA2
					A	2,DATA2
					A	3,DATA2
	⋮					
DATA1	DC	F'5'		DATA1	DC	F'5'
DATA2	DC	F'10'		DATA2	DC	F'10'
	⋮					

マクロ呼出しには1個以上の引数を使用することができる．各引数はマクロ定義中のマクロ名の行に示された定義用引数(“仮引数”)と対応しなければならない．マクロ呼出しが処理されるとき，与えられた引数はマクロ定義の中の対応する仮引数に代入される．

次のプログラムを考えてみよう．

例 3：

```
            ⋮
LOOP1  A    1, DATA1
       A    2, DATA2
       A    3, DATA3
```

```
                    ⋮
LOOP2   A           1, DATA3
        A           2, DATA2
        A           3, DATA1
                    ⋮
DATA1   DC          F'5'
DATA2   DC          F'10'
DATA3   DC          F'15'
```

この場合には同じ命令の並びでも演算数と最初のカードのラベルとはそれぞれ異なっている．このプログラムは次のように書くことができる．

```
            原始プログラム                    |      展開された原始プログラム
                ⋮                             |
        MACRO                                 |
&LAB    INCR        &ARG1,&ARG2,&ARG3         |
&LAB    A           1,&ARG1                   |
        A           2,&ARG2                   |
        A           3,&ARG3                   |
        MEND                                  |
                ⋮                             |              ⋮
LOOP1   INCR        DATA1,DATA2,DATA3         | { LOOP1   A         1,DATA1
                                              | {         A         2,DATA2
                                              | {         A         3,DATA3
                ⋮                             |              ⋮
LOOP2   INCR        DATA3,DATA2,DATA1         | { LOOP2   A         1,DATA3
                                              | {         A         2,DATA2
                                              | {         A         3,DATA1
                ⋮                             |              ⋮
DATA1   DC          F'5'                      |   DATA1   DC        F'5'
DATA2   DC          F'10'                     |   DATA2   DC        F'10'
DATA3   DC          F'15'                     |   DATA3   DC        F'15'
                ⋮                             |
```

上では**ラベル引数**を含めた4つの引数を定義している．ラベル引数はマクロ命令の“演算数”として記されているものと全く同様に処理される．マクロ名の行とマクロ呼出しのときにラベル部に現われる引数は，他の引数と書かれる

場所が違うだけである．すなわち，マクロの中でラベルとして使用される引数は，呼出しの時必らずしもラベル部に書かなくてもよい．また，呼出しのときラベル部に書いた引数は，マクロ命令コードの中で演算数として使用することもできる．ラベル引数は，マクロ命令と通常のアセンブラ語命令との類似性を強調するために具合が良いものであるが，なければならぬというものではない．次は例3を別の方法でコーティングしたものである．

原始プログラム	展開された原始プログラム
<pre> MACRO INCR &ARG1,&ARG2,&ARG3,&LAB &LAB A 1,&ARG1 A 2,&ARG2 A 3,&ARG3 MEND . . . INCR DATA1,DATA2,DATA3,LOOP1 . . . INCR DATA3,DATA2,DATA1,LOOP2 . . .</pre>	<pre> ⋮ LOOP1 A 1,DATA1 A 2,DATA2 A 3,DATA3 ⋮ LOOP2 A 1,DATA3 A 2,DATA2 A 3,DATA1 ⋮</pre>

マクロ呼出しの引数を指定する方法には一般に2つある．第1の**位置決め引数**は前述の例で示されている．引数と仮引数は現われる順番に対応付けられる．したがって，マクロ呼出し‘INCR　A, B, C’においてA, B, Cは各々第1，第2，第3の仮引数と置換えられる．もっと一般的な引数である**見出し語付き引数**は，仮引数を位置でも名前でも参照できるようにするものである．INCRの定義で現われる仮引数を参照するために次のように書くことができる．

　　　‘INCR　&ARG 1=A , &ARG 3=C , &ARG 2=B’ または

　　　‘INCR　&ARG 1=A , &ARG 2=B , &ARG 3=C’

この機能はつねには必要としない引数，例えばラベル，を持っているとき有効である．与えられなかった引数はマクロ・プロセッサによって空白とされる．

4.2.2 条件付きマクロ展開

2つの重要なマクロ・プロセッサに対する擬似命令 AIF と AGO を用いると，条件によってマクロ展開を制御することができる．これによりマクロ呼出しの展開で作られる機械語命令を条件付きで選択することが可能となる．

次のプログラムを考えてみよう．

例　4：

```
               ⋮
LOOP1   A            1, DATA1
        A            2, DATA2
        A            3, DATA3
               ⋮
LOOP2   A            1, DATA3
        A            2, DATA2
               ⋮
LOOP3   A            1, DATA1
               ⋮
DATA1   DC           F'5'
DATA2   DC           F'10'
DATA3   DC           F'15'
               ⋮
```

この例では，演算数，ラベルおよび生成される命令の個数は各並びによって異なっている．このプログラムは次ページのようにして書くことができる．

ピリオド“.”で始まるラベル，例えば .FINI は**マクロ・ラベル**であり，マクロ・プロセッサの出力には現われない．文 AIF (&COUNT EQ 1) .FINI は，もし &COUNT に対応するパラメータが1であれば，マクロ・プロセッサは .FINI というラベルまで飛び，1でなければ AIF 擬似命令の次の文を続けて処理することを指定する．

AIF は**条件分岐**擬似命令であり，算術演算の結果を判定し条件が真であれば分岐を行なう．AGO は**無条件分岐**擬似命令，または 'GO TO' 文である．この命令はマクロ命令定義の中の他の文に付けられたラベルを指定し，マクロ・プロセッサはその命令へ行って処理を続ける．これらの文はマクロ・プロセ

```
             ・
             ・
             ・
             MACRO
&ARG0        VARY         &COUNT,&ARG1,&ARG2,&ARG3
&ARG0        A            1,&ARG1
             AIF          (&COUNT EQ 1).FINI           & COUNT = 1 の判定
             A            2,&ARG2
             AIF          (&COUNT EQ 2).FINI           & COUNT = 2 の判定
             A            3,&ARG3
.FINI        MEND
             ・
             ・
             ・
LOOP1        VARY         3,DATA1,DATA2,DATA3
             ・
             ・
             ・
LOOP2        VARY         2,DATA3,DATA2
             ・
             ・
             ・
LOOP3        VARY         1,DATA1
             ・
             ・
             ・
DATA1        DC           F'5'
DATA2        DC           F'10'
DATA3        DC           F'15'
```

展開された原始プログラム

```
            ・
            ・
            ・
LOOP1       A        1,DATA1
            A        2,DATA2
            A        3,DATA3
            ・
            ・
            ・
LOOP2       A        1,DATA3
            A        2,DATA2
            ・
            ・
            ・
LOOP3       A        1,DATA1
            ・
            ・
            ・
```

ッサに対する指令であって，マクロ展開の結果には現われない．ちょうど機械語プログラムの条件付き，および無条件分岐命令がプログラムの流れを変更するように，AIF および AGO は，マクロ・プロセッサがマクロ命令の文を展開するときにその順序を制御する．これによって場合場合に応じて特別の命令の列を仕立てることが可能となる．マクロの条件付き展開ができないときは実行プログラムの中で判定や分岐が必要となる．もし機械語の分岐や判定命令が使用されると実行時間を浪費するし，判定命令や使用されない命令は記憶装置をも無駄に使用する．マクロ言語の分岐と判定によって，各呼出しのパラメータを判定して不要なコードを省略し，選択的にコードを組立てるような高度な汎用マクロを使用することができる．この高度な汎用マクロの選択的展開機能は，システム・プログラマの最も強力な道具である．

4.2.3 マクロの中でのマクロ呼出し

マクロ呼出しは一連の命令の“省略形”であるから，他のマクロ定義の中でこのような“省略形”が使用できるようにすることが望ましいと考えられる．

例　5：

```
MACRO
ADD1      &ARG
L         1, &ARG
A         1, =F'1'
ST        1, &ARG
MEND
MACRO
ADDS      &ARG1, &ARG2, &ARG3
ADD1      &ARG1
ADD1      &ARG2
ADD1      &ARG3
MEND
```

マクロ‘ADDS’の定義の中では前に定義されているマクロ‘ADD1’が3回呼出されている．マクロ‘ADD1’を使用することによって，‘ADDS’の定義の長さが短くなり，より容易に理解できる．

このようにマクロを使用するとマクロ展開で複数の‘レベル’ができる．

原始プログラム	展開された原始プログラム（レベル 1）	展開された原始プログラム（レベル 2）
.		
.		
.		
MACRO		
ADD1　&ARG		
L　1,&ARG		
A　1,=F'1'		
ST　1,&ARG		
MEND		
MACRO		
ADDS　&ARG1,&ARG2,&ARG3		
ADD1　&ARG1		
ADD1　&ARG2		
ADD1　&ARG3	ADDSの展開	ADD1の展開
MEND	.	.
	.	.

```
   .                          |                    |       {L    1,DATA1
   .                          |  {ADD1  DATA1      |       {A    1,=F'1'
   .                          |  {                 |       {ST   1,DATA1
                              |  {                 |       {L    1,DATA2
        ADDS   DATA1,DATA2,   |  {ADD1  DATA2      |       {A    1,=F'1'
               DATA3          |  {                 |       {ST   1,DATA2
                              |  {                 |       {L    1,DATA3
                              |  {ADD1  DATA3      |       {A    1,=F'1'
                              |  {                 |       {ST   1,DATA3
   .                          |                    |        .
   .                          |                    |        .
   .                          |                    |        .
DATA1  DC    F'5'             |                    |DATA1  DC    F'5'
DATA2  DC    F'10'            |                    |DATA2  DC    F'10'
DATA3  DC    F'15'            |                    |DATA3  DC    F'15'
   .                          |                    |        .
   .                          |                    |        .
   .                          |                    |        .
```

マクロの中でマクロ呼出しを行なうと，展開のために幾つかのレベルが生じる．例えば，マクロ ADDS は他のマクロ定義の中で呼出されるかも知れない．実際に，条件付きマクロの機能（AIF や AGO など）は自分自身を呼ぶことも可能とする．これが無限ループとならない限り，*n* 回目には自分自身を呼ばないと判定するような場合，これは意味のあることである．マクロの中でのマクロ呼出しに固有な問題は**回帰呼出し**（recursion）の節で論じる．前述の例は**図 4.5** でも用いられる．

4.2.4 マクロを定義するマクロ命令

これまでマクロは一連の命令の一般的な省略形であると考えてきた．そしてマクロ命令の定義をも含め，省略形の中で有効な文を書けるようにすることが望ましいことを示した．このような機能があると類似したマクロ群の定義を1つのマクロ定義によって簡単に行なうことができる．

この場合には，内側のマクロ定義は，外側のマクロが呼出されて初めて定義される（すなわち，呼出し可である）ようにしておくことが重要である．この理由は，定義が利用される方法から生じる．例えば，標準の呼出し手順を用いてサブルーチンを呼出すマクロ群を定義してみよう．次の例はマクロ命令 DEFINE を定義している．DEFINE は，サブルーチン名を伴って呼出されたとき，この名前をサブルーチン名としてマクロを定義する．生成された各マク

ロの名前は（引数 &SUB によって与えられた）サブルーチン名である．このような場合には内側のマクロ定義は外側のマクロが呼出されて初めて定義されることに意味がある．

マクロ DEFINE の定義 ｛ マクロ &SUB の定義 ｛

```
MACRO
DEFINE  &SUB            マクロ名 : DEFINE
MACRO
&SUB    &Y              仮引数のマクロ名
CNOP    0,4             語境界の調整
BAL     1,*+8           パラメータ表ポインタをレジスタ1に設定
DC      A(&Y)           パラメータ表ポインタ
L       15,=V(&SUB)     サブルーチンの番地
BALR    14,15           サブルーチンに制御を移す
MEND
MEND
```

利用者はこのマクロを次のようにして呼出すことができる．

例　6：

```
DEFINE    COS
```

この文は COS という名前の新しいマクロを定義し，新しいマクロ定義に展開される．その後に，利用者は COS マクロを次のように呼出すことができる．

```
COS       AR
```

マクロ・プロセッサは次の呼出し手順を生成する：

```
BAL     1,*+8
DC      A(AR)           ARの番地
L       15,=V(COS)      V は外部記号の番地を示す
BALR    14,15
```

マクロの中でマクロ定義はしばしば“マクロ定義の中でのマクロ定義”ともいわれる．（この例は第 4.3.2 節でさらに展開される．）

4.3 組　込　み

これまでマクロ命令を基本的なアセンブラ語を拡張する目的で述べてきた．本章の残りの部分はマクロのインプリメンテーションを概説する．

問題の設定

マクロ命令プロセッサが果さなければならない役割りは4つある．

1. マクロ定義の認識　マクロ命令プロセッサは MACRO および MEND 擬似命令で示されたマクロ定義を認識しなければならない．マクロの中にマクロ定義が現われるときは複雑になる可能性がある．前節の例のように，MACRO と MEND が**入れ子**になっているとき，マクロ・プロセッサは入れ子の関係を正しく識別し，MACRO と MEND の対応を付けなければならない．入れ子になった MACRO と MEND を含め，それらの間にある一群の命令は全体で1つのマクロ命令を定義する．

2. 定義の保存　プロセッサはマクロ命令の定義を記憶しなければならない．これはマクロ呼出しを展開するのに必要である．

3. 呼出しの認識　プロセッサは命令記号として現われるマクロ呼出しを認識しなければならない．

4. 呼出しの展開と引数の代入　プロセッサは仮引数，すなわち，マクロ定義の引数に対応したマクロ呼出しの引数を代入しなければならない．こうしてできた命令列（この場合アセンブラ語）はマクロ呼出しの所へ代入される．もちろん，この命令群はさらにマクロ定義や呼出しを含むことができる．

要　点：　マクロ・プロセッサはマクロ定義とマクロ呼出しを認識し処理しなければならない．

引数に関してマクロ・プロセッサの設計者は幾つかの決定をしておかなければならない．マクロ定義のどこに仮引数を認めるか，例えば，命令コードとして使用してよいかなどである．また，使用可能な引数の構文も定めておかなければならない．ここではこれらの詳細を述べないが，幾つかの典型的な（または合理的な）例を示す．仮引数はマクロ定義の**任意**の場所に用いてもよい．例えば，命令コードとして用いてもよい．また，マクロ引数と固定の記号列とを結合することも可能としたい．例えば，A&A と書いてAと，&A の所に代入される引数とを結合したい．このために引数を“ ′ ”で囲んで示すことも認める．

したがって，A&AB はAと，&ABに代入される引数とを結合することを，A'&A'B はAと，&A に代入される引数と，B とを結合することを示す．かっこで囲まれた記号列（例えば，(. A&B)）も引数として使用してよい．

AIF および AGO が正しく働くためには算術式の計算が必要である．EQU 命令によって変数に値を割付けたり変更したりできるのと同様に，記号の定義を行なえる機能がしばしば必要となる．

4.3.1 限定された機能の組込み：2パス・アルゴリズム

まず簡単な仮定を行なう．マクロ・プロセッサは機能的にアセンブラとは独立であり，マクロ・プロセッサの出力テキストがアセンブラの入力とされる．最初はマクロ定義の中でのマクロ呼出しや定義を禁止しておく．この理由は論理が複雑になるからであり，基本的なマクロ・プロセッサのアルゴリズムを述べるまでこの議論を行なわない．

マクロ・プロセッサはアセンブラと同様に原始プログラムの行を走査し，処理を行なう．アセンブラ語の場合には行は番地と関係があった．1つの行は番地やアセンブラに“既知”の名前によって他の行を参照することができる．さらに各行に割付けられる番地は先行する行の番地や内容に関係する．マクロ定義が1つのかたまりを構成すると考えるとき，マクロ言語の行はさほど密接に関連していないといえる．マクロ定義はその外側のものを参照することはなく，マクロ呼出しはマクロ定義のみを参照する．（前述の制限事項，および，マクロ呼出しは値でなく原始文を代入することに注意．例えば，マクロ呼出し INCR Y の前にYを定義する文，例えば Y EQU 10 があったとしても，マクロ・プロセッサは‘10’ではなく‘Y’をマクロ定義の引数に代入する．マクロ・プロセッサは原始文の編集をする以外 EQU 文を処理しない．）

マクロ・プロセッサは解釈のために原始文を2回走査（パス）する．すなわち，1回目はマクロ定義を探して，2回目はマクロ呼出しを探す．アセンブラで，定義されるよりも前に記号の参照を処理できなかったのと全く同様に，マクロ・プロセッサでも，マクロ定義を見付けそれを記憶するまではマクロ呼出しを展開することはできない．したがって，入力デックを2回のパスで調べることが必要である．1回目は定義を処理し，2回目は呼出しを処理する．パス

1はすべての命令コードを調べて，すべてのマクロ定義をマクロ定義表（MDT）に保存する．また，パス2で使用するために入力デックからマクロ定義を除いたものを2次記憶装置（例えば，磁気テープ）に書込んでおく．パス1は MDT と共にマクロ名表（MNT）をも作る．次にパス2はすべての命令記号を調べ，マクロ名に対してマクロの定義から適当な一群の命令を作り出して置換える．（MNT は必らずしも別個に設けなくてもよいが，別にしておくとマクロ名の探索が容易である．）

データ・ベースの規定

次のデータ・ベースがマクロ・プロセッサの2つのパスによって使用される．

パス1のデータベース：

1. 入力——マクロ原始デック．
2. 出力——マクロ原始デックの写し．パス2で使用する．
3. マクロ定義表(MDT)——マクロ定義の本体を格納するために使用する．
4. マクロ名表（MNT)——定義されたマクロの名前を格納するために使用する
5. マクロ定義表カウンタ（MDTC)——次に使用できる MDT の欄を示す．
6. マクロ名表カウンタ（MNTC)——次に使用できる MDT の欄を示す．
7. 引数表配列（ALA)——マクロ定義を格納するときに仮引数に対して指標を代入するために使用する．

パス2のデータ・ベース：

1. 入力——マクロ原始デックの写し．
2. 出力——アセンブラの入力として使用される展開された原始デック．
3. マクロ定義表（MDT)——パス1で作られた．
4. マクロ名表（MNT)——パス1で作られた．
5. マクロ定義表ポインタ（MDTP)——マクロ展開の間使用される原始文の次の行を指す．
6. 引数表配列（ALA)——マクロ定義の中に，指標の代りにマクロ呼出しの引数を代入するために使用する

データ・ベースの書式の規定

マクロ定義表（MDT），マクロ名表（MNT）および引数表配列（ALA）の詳細のみを示す．他は簡単であるので省略する．

引数表配列：引数表配列（ALA）はパス1とパス2の両方で使用されるが幾分逆の機能を持ったものとして使用される．パス1では，マクロ展開のとき後で出てくる引数の置換えを簡単にするために，マクロ定義の中の仮引数は定義が格納されるときにその位置を示す指標で置換えられる．マクロ名カードの i 番目の引数はマクロの本体の中で指標記号 #i を用いて示されている．ただし“#”はマクロ・プロセッサのためにとってある記号である（すなわち，プログラムは使用できない）．これらの記号は，マクロ呼出し展開のときにはあらかじめ用意された引数表と共に使用される．仮引数の記号名はマクロ名カードに保存され，引数の置換えが位置でなく名前ででもできるようにしている．

例3で使用された INCR マクロを考えてみよう．格納されたマクロ定義は

```
        マクロ定義表MDT
             ⋮
&LAB    INCR     &ARG1,&ARG2,&ARG3
#0      A        1, #1
        A        2, #2
        A        3, #3
        MEND
             ⋮
```

パス2において，マクロ定義の中に記入された指標をマクロ呼出しの引数で置換えなければならない．したがって，次の呼出しが行なわれると

```
LOOP    INCR    DATA1,DATA2,DATA3
```

マクロ呼出し展開ルーチンは次の引数表配列を用意する：

引数表配列	
	各欄 8 バイト
指標	引数
0	"LOOP1*bbb*"
1	"DATA1*bbb*"
2	"DATA2*bbb*"
3	"DATA3*bbb*"

（*b* は空白を示す）

この表はこの呼出しの展開の期間だけで使用される．さらに次のような呼出しがなされたと考える：

INCR　　&ARG1=DATA3, &ARG2=DATA2, &ARG3=DATA1

マクロ・プロセッサは '&ARG1', '&ARG2', '&ARG3', がマクロ名カード上で引数の 1, 2, 3 番目に対応することを知る．この結果引数表配列は次のようになる：

引数表配列		
指標	引数	
0	"*bbbbbbbb*"	（全て空白）
1	"DATA3*bbb*"	
2	"DATA2*bbb*"	
3	"DATA1*bbb*"	

マクロ定義表：　マクロ定義表 (MDT) はテキスト行の表である．入力が 80 欄カードのとき，MDT は 80 バイトの記号列の欄を持つ表とされる．MACRO 行以外のマクロ定義のすべての行は MDT に格納される．(MACRO 行はマクロ展開に際して使用されない.) MEND は定義の終りを示すために残される．マクロ名の行は見出し語による引数置換えを可能とするように保存される．したがって，例の INCR マクロは次のように格納される：

マクロ定義表			
			各欄 80 バイト
指標			カード
⋮		⋮	
15	&LAB	INCR	&ARG1, &ARG2, &ARG3
16	#0	A	1,#1
17		A	2,#2
18		A	3,#3
19		MEND	
⋮		⋮	

マクロ名表：　マクロ名表は，アセンブラの機械命令表（MOT）および擬似命令表（POT）と非常によく似た機能を果す．MNT の各欄は，文字列（マクロ名），および，マクロ定義の開始位置に対応する MDT の欄へのポインタから成っている．MNT の INCR マクロに対する欄は次のようになっている：

	8バイト	4バイト
指標	名前	*MDT* 指標
⋮	⋮	⋮
3	"INCR*bbbb*"	15
⋮	⋮	⋮

アルゴリズム

図 4.1 および **図 4.2** はマクロ定義と展開のアルゴリズムを示した流れ図である．カード読込みの箱は2次記憶装置から作業領域へ次の行を取り出すことを示している．

パス1——マクロ定義：　パス1のアルゴリズム（**図 4.1**）は各入力行を判定する．もし MACRO 擬似命令であれば，引続くマクロ定義全体はマクロ定義表（MDT）の次に空いている場所に保存される．定義の第1行はマクロ名の行である．名前はマクロ表（MNT）に，定義を格納した MDT の先頭位置を示すポインタと共に登録される．END 擬似命令が見付かるとすべてのマクロ定義は処理されたものとして，引続いてマクロ呼出しを処理するために制御はパス2に移される．

パス2——マクロ呼出しと展開：　パス2のアルゴリズム（**図 4.2**）は各入力行の命令記号が MNT に入っているか否かを調べる．呼出しが見付かると呼出し処理ルーチンはマクロ定義表ポインタ（MDTP）を，MDT の中の対応したマクロ定義に設定する．MDTP の初期値は MNT の欄の"MDT 指標"部から求められる．マクロ展開ルーチンは，仮引数の指数とそれに対応した呼出しの引数とから成る引数表配列（ALA）を用意する．この表は，呼出しの引数と名前カードに記された仮引数との対応をとるために，マクロ呼出しの引数とし

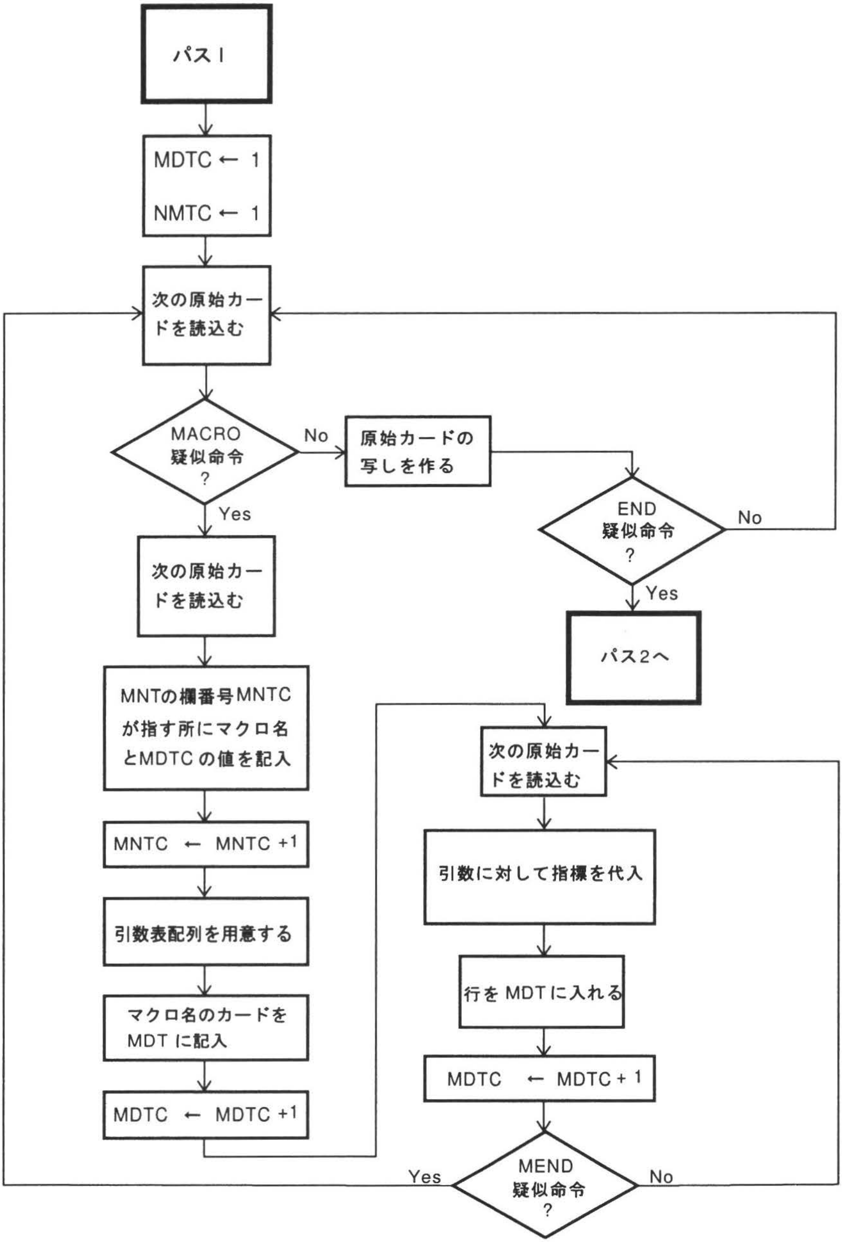


図 4.1 パス 1——マクロ定義の処理

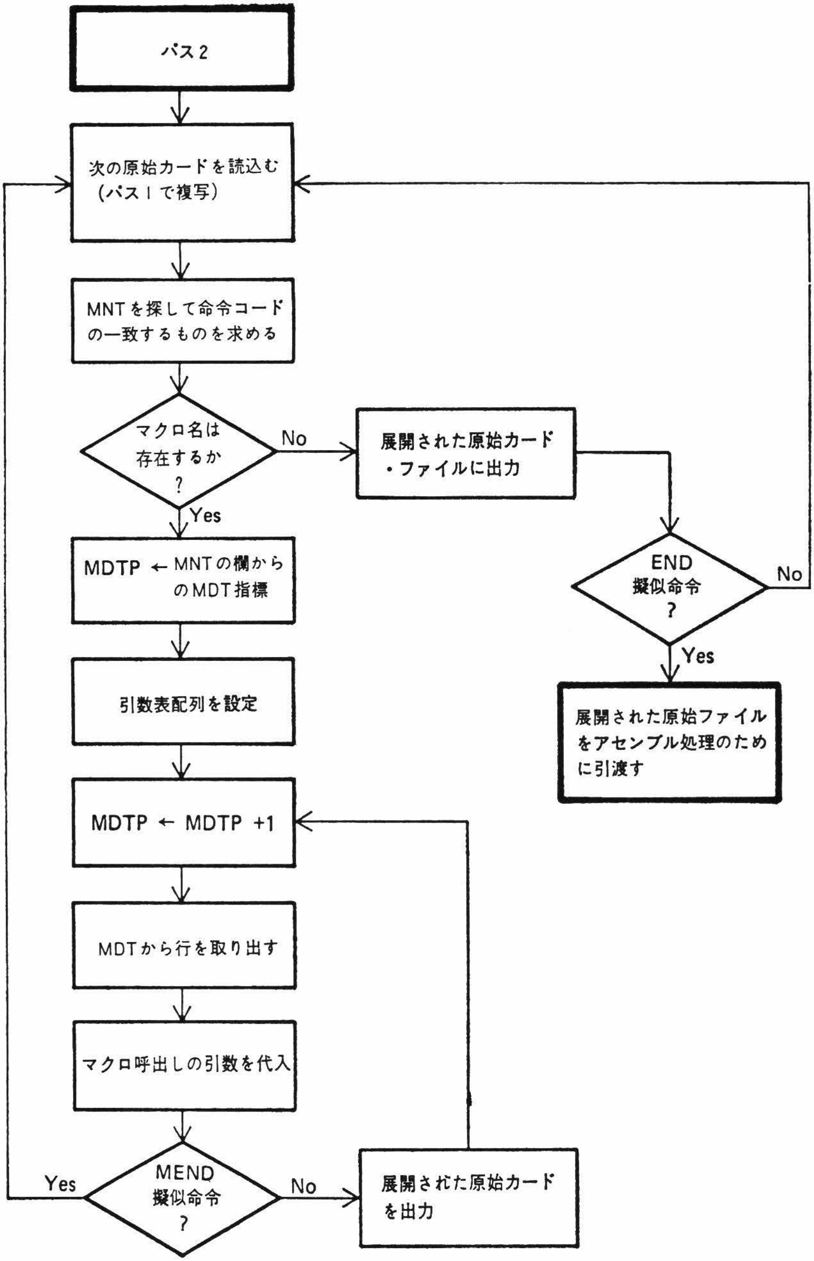


図 4.2　パス2——マクロ呼出しと展開の処理

て用いられた記号を順に並べたものである（第1はラベル引数で指数0を持っている）．呼出しで指定されなかった引数は空白とみなし，また，余分の引数は無視する．位置による引数参照の場合には，この対応付けは全く明解である．名前による参照に対しては，マクロ・プロセッサは，マクロ名の行（これは MDT の定義の始めにある）で仮引数を探して適当な指数を決定する．

MDT からの読込みが続けられ，各行が読込まれると，マクロ定義の中の仮引数の指数に対して引数表からの値が代入される．MDT から MEND を読むとマクロの展開が終了する．そして入力ファイルにおいて走査が続けられる．END 擬似命令がくると，展開された原始デックはアセンブラに渡されて次の処理が行なわれる．

4.3.2 1パスのアルゴリズム

マクロの中でマクロ定義を行なうこととしょう．基本的な問題は，内側のマクロは外側のマクロが呼出されてから定義されるということである．内側のマクロを使用するためには，マクロ定義とマクロ呼出しのパスを繰り返さなければならない．しかし，すべてのマクロ処理を1パスにすることにより，これを簡単に解決できると同時にある種の利点も生じる．再びアセンブラに類似点を求めてみよう．マクロは，その呼出しを展開できるようになる前に処理ルーチンに対して定義されていなければならないので，マクロ定義は呼出しの前に処理されなければならない．すべてのマクロは呼出される前に定義されていなければならないという制限を明記すると，1パス処理に対する障害を取り除くことができる．（記号に対する同じような要求は，プログラムの流れを不当に制限することにはなるが，アセンブラでも成立する．）マクロ言語の場合に，すべてのマクロ定義は呼出しの前に置くという要求は不合理ではない．この要求はマクロの使用に対して重大な制限を課すことにならない．マクロ定義の中のマクロ呼出しはその展開がマクロの定義の期間にではなく呼出されたときに行なわれるので，この要求はマクロが自分自身を呼ぶことすら防げるものではない．

図4.3 および **図4.4** の流れ図は **図4.1** および **図4.2** のアルゴリズムを1つのパスに結合したものであり，マクロの中でのマクロの定義も可能である（ただし，マクロの中でのマクロ呼出しはできない）．これらの新しい流れ図と

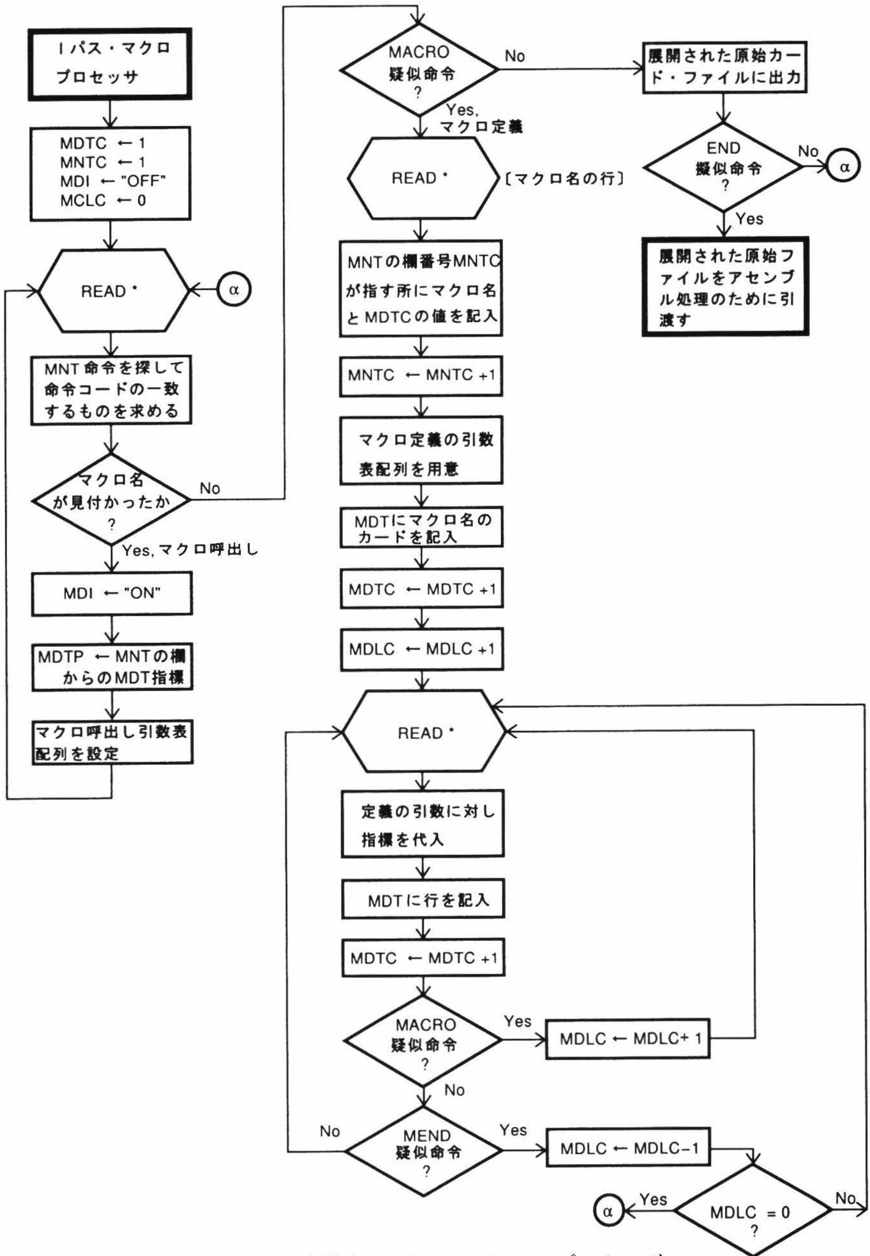


図 4.3　簡単な 1 パス・マクロ・プロセッサ

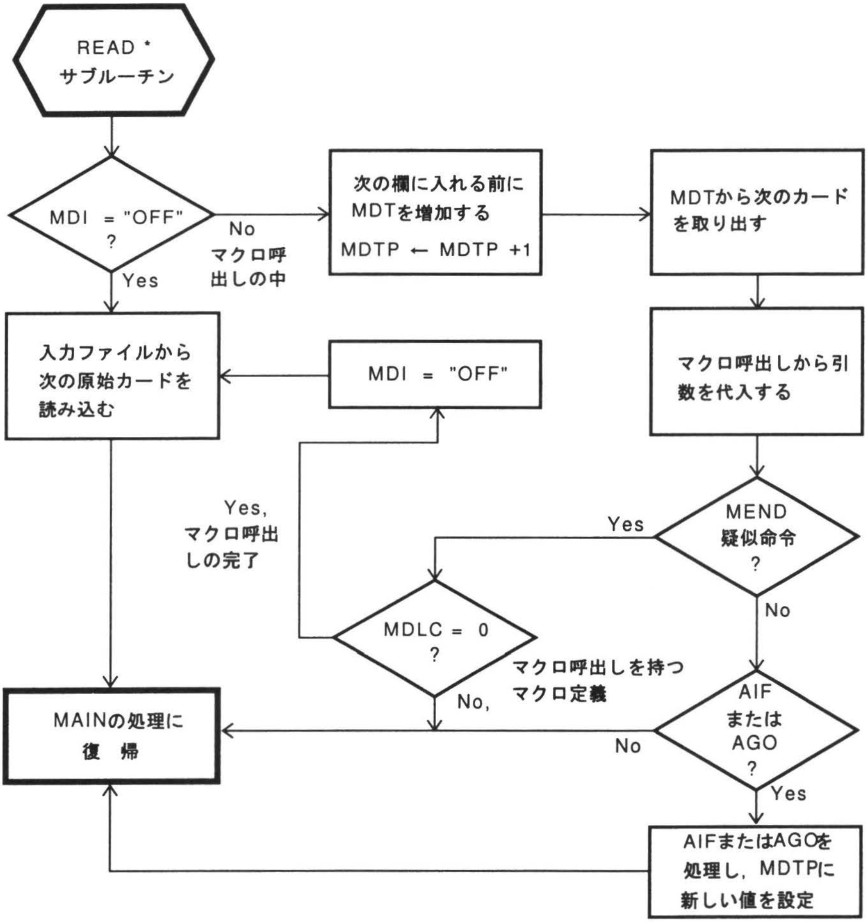


図 4.4 マクロ展開に使用される READ 機能の詳細

前の 2 パス・マクロ・プロセッサの流れ図は非常によく似ていることに注意してほしい．1 パスの設計には，マクロ定義入力 (MDI) 表示子およびマクロ定義レベル・カウンタ (MDLC) の 2 つの変数が追加されている．MDI と MDLC はマクロ呼出しとマクロ定義を記憶しておくためのスイッチ（カウンタ）である．

MDI 表示子はマクロ呼出しの期間 *"on"* の値を持ち，他の期間 *"off"* の値を持つ．実際のマクロ展開は READ の箱で行なわれる．この詳細は 2 番目の

流れ図（**図 4.4**）に示されている．READ はスイッチ MDI を調べ *on* であれば行はマクロ定義表（MDT）から読込まれる．マクロの終了を示す MEND 行を読込むと呼出しの展開を終了する．MDI は *off* にされ，次の行は正規の入力流から求められる．注目すべきことは READ によって返却される行はマクロ定義を持っていてもよく，展開されたマクロ・コードは他のコードと全く同じように READ から出てくることである．したがって，展開されたコードは新しいマクロ定義をも含み得る．

MACRO 擬似命令がくるとマクロ定義レベル・カウンタは1だけ増され，MEND 擬似命令がくると1だけ減ぜられる．

MDLC は MACRO および MEND を含めたマクロ定義全体が MDT に正しく格納されていることを確認するために使用される．これは入れ子になったかっこ，例えば“(2*(A+B)+C)”，を持つ文の処理と類似している．プログラムで式を全部読込むためには，第1の右かっこで停止してはならず，左と右のかっこの個数を数えて第2のかっこが式の真の終りであることを認識しなければならない．MDLC は出てきた MACRO と MEND の個数の差を数えるカウンタとして働く．すなわち，MDLC の値は MEND より MACRO がどれだけ多いかを示す．

“マクロ定義のパス”と“マクロ展開のパス”を同時に実行することが可能であるためには，中にマクロの定義を持つマクロ定義を展開するとき，2つの別の引数表配列が必要である．1つの ALA はマクロ定義のために使用され，他はマクロ呼出し展開のために使用される．

次に，第 4.2.4 節の例6を考えてみよう．マクロの定義 DEFINE の後に：

MDT:

指数		
1	DEFINE	&SUB
2	MACRO	
3	#1	&Y
4	CNOP	0,4
5	BAL	1,*+8
6	DC	A(&Y)
7	L	15,=V(#1)
8	BALR	14,15
9	MEND	
10	MEND	

MNT:

指数	名前	MDT 指数
1	DEFINE	1

文 DEFINE COS を処理した後に：

MDT:

	行 1-10 は上と同じ	
.	.	
.	.	
.	.	
11	COS	&Y
12	CNOP	0,4
13	BAL	1,*+8
14	DC	A(#1)
15	L	15,=V(COS)
16	BALR	14,15
17	MEND	

MNT:

指数	名 前	MDT 指数
1	DEFINE	1
2	COS	11

これまでにはマクロによって何も出力されていない．出力は文 COS AR が展開されたとき生成される．読者は前述の例に対して流れ図を追い，マクロ・プロセッサの動作を“模擬（シミュレート）”してみてほしい．

4.3.3 マクロの中でのマクロ呼出しの組込み

マクロの中でのマクロ呼出しを組込むための基本的な問題点は回帰呼出しである．マクロの展開中にマクロの呼出しが起こると，マクロ・プロセッサは内側のマクロ呼出しを展開してから，外側のマクロの展開を行なわねばならない．2回目の呼出しは第2のマクロ・プロセッサによって展開される．第2のマクロ・プロセッサは，マクロ定義を MDT の中で探して展開したコードを第1のマクロ・プロセッサに返却する．たくさんのマクロ・プロセッサを持つことは効率も悪いし一般性もない．しかし，1つのマクロ・プロセッサで入れ子のマクロ呼出しを処理することとなると，何らかの方法でプログラムの状態を保存しなければならない．

マクロ定義表*MDT*

指標	内容	
1	ADD1	&ARG
2	L	1, #1
3	A	1,=F'1'
4	ST	1, #1
5	MEND	
6	ADDS	&ARG1,&ARG2,&ARG3
7	ADD1	#1
8	ADD1	#2
9	ADD1	#3
10	MEND	

図 4.5　例5のマクロ定義後のMDT

例5はこの問題を示している．例は2つのマクロ ADDS と ADD1 を定義している．MDT の中のマクロ定義は**図 4.5**に示されている．

マクロ呼出し

```
ADDS        DATA1, DATA2, DATA3
```

が出てきたときの**図 4.3**および**図 4.4**のマクロ・プロセッサの動作を考えてみよう．このアルゴリズムはマクロ呼出し引数表配列を作り，ポインタ MDTP を MDT の行6に設定し，スイッチ MDI を*on*にする．READ 関数は MDTP を1だけ増し，MDT から次の行（行7）を求め，引数を代入して新しい行を作る．

```
ADD1        DATA1
```

ここでこのアルゴリズムは混乱に陥る．他のマクロ呼出しが起こったのである．ADD1 はマクロ名であり，プロセッサは新しい引数表配列を作り，MDTP を MDT の行1に設定し，MDI を *on*（同じ値）とする．ADD1 マクロは正しく展開されるであろう．そして終了時に MDTP は MDT の行5を指している．MDI は *off* にされ，読込みは入力流から続けられる．

これはもちろん正しくない．ADDS マクロの中で展開されずに残ってしまった行がある．3つの誤りが生じたのである．スイッチ MDI は*off*にされ，プロセッサの読込みは MDT からではなく入力流からとなった．ADD1 マクロ

を処理するため MDTP を設定し直したので，ADDS を展開するための MDTP の値は失われてしまっている．また，ADDS を呼出すための引数表も失われている．

マクロの中でのマクロ呼出しを処理するためにマクロ・プロセッサは回帰的に動作可能でなければならない．すなわち，1つの処理がすむ前に次のマクロの処理を行ない，終了すれば前の‘外側’のマクロの処理を続ける．回帰手続きは通常**スタック**を用いて実行される．スタックとは，手続きに対しての各呼出しに関連する変数のために，別の記憶領域を割当てる記憶方法である．独立の記憶領域，**スタック・フレーム**，は回帰呼出しに対応しているので，未完の計算の状態は保存される．概念的にスタックは無限長の LIFO（Last In First Out）によって処理される配列である．最後に格納された要素が最初に取り出される．**スタック・ポインタ**（SP）はスタックの位置，すなわち，**フレーム**の開始点，を指す．（**ポインタ**とは番地や指標として用いられるデータ項目をいう．直観的に，ポインタはコア中に格納された情報を指す．）

例では各呼出しに対するポインタ MDTP と引数表 ALA はスタックに格納される．スイッチ MDI も同様に保存されなければならない．現在のスタック・フレームの開始点を指すスタック・ポインタ（SP）は MDI の代りに使用できる．後に続くスタック・フレームの中で以前の SP の値を保存して，この値を失わないようにしなければならない．（この方法とマクロ定義レベル・カウンタ（MDLC）の使用法とを比較せよ．マクロ定義の‘回帰的’処理は，内側の定義がそれに関したデータを持たなかったので容易であった．本章末の演習問題参照．）実際のデータ表現法を無視して，スタックがポインタと文字列との配列から構成されていると考える．S(SP) は現在のスタック・フレームの開始点 SP によって指される位置を参照するとする．

各スタック・フレームの構成は**図 4.6** に示されている．S(SP) は SP の**以前の**値を納めている．第1フレーム SP=1 に対して，S(SP)=−1 とする．条件 SP=−1 は，非回帰的アルゴリズムにおける MDI が *off* の場合と同様に，マクロ・プロセッサがマクロ呼出し展開を行なっていないことを示す．S(SP+1)は現在の MDTP，MDT ポインタ，の値を納めている．S(SP+2)……S(SP+N+1) は現在の引数表の内容としてN個の文字列を納めている．

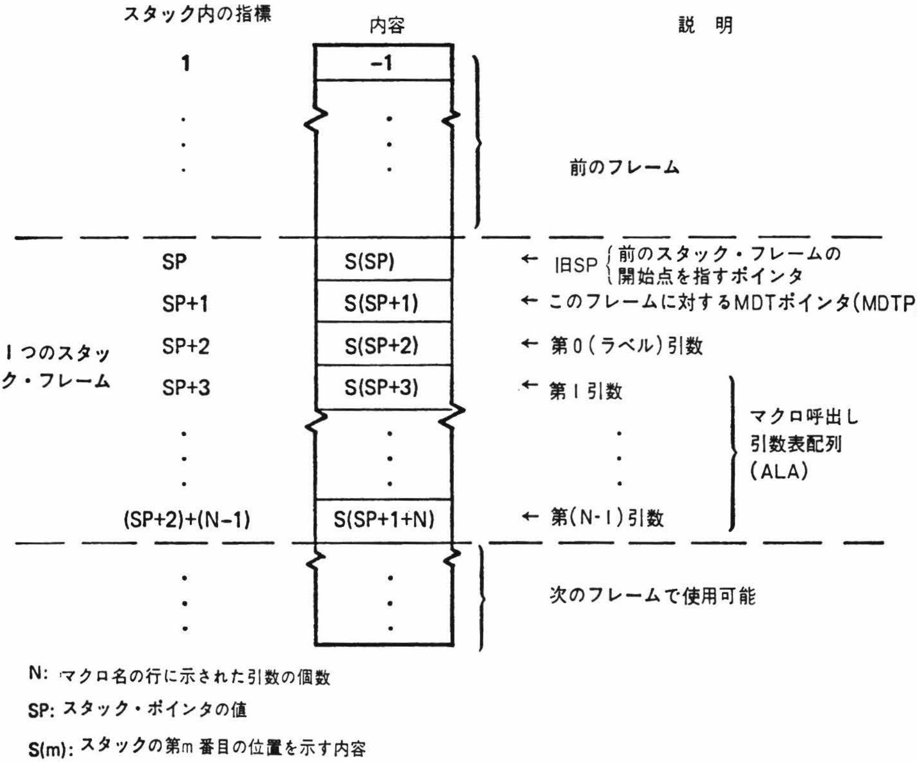


図 4.6　スタックの構成

図4.7および**図4.8**は，マクロ定義の中でのマクロ定義の処理，および，マクロ定義の中でのマクロ呼出しの処理を行なうマクロ・プロセッサのアルゴリズムを示している．新しい欄を挿入した後のマクロ名表カウンタ（MNTC）やマクロ定義表カウンタ（MDTC）の増加などの簡単な“雑用”のステップは流れ図を簡潔にするために省略してある．読者はこれらの新しい流れ図で示されているアルゴリズムが前のアルゴリズム（**図4.3**および**図4.4**）ときわめてよく似ていることを確かめてほしい．なお，スタック・フレームの欄数は引数表の要素の個数（すなわち，マクロ定義の仮引数の個数）に依存する．

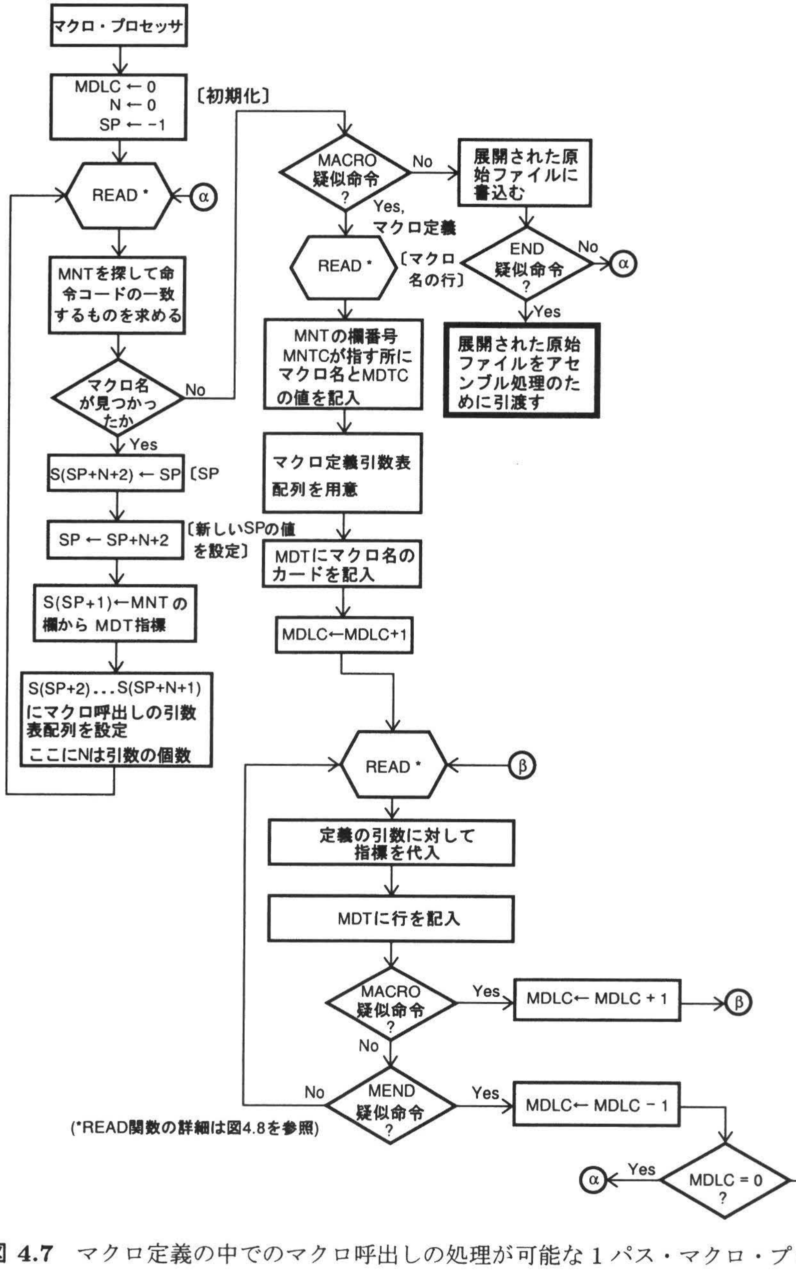


図 4.7 マクロ定義の中でのマクロ呼出しの処理が可能な 1 パス・マクロ・プロセッサ

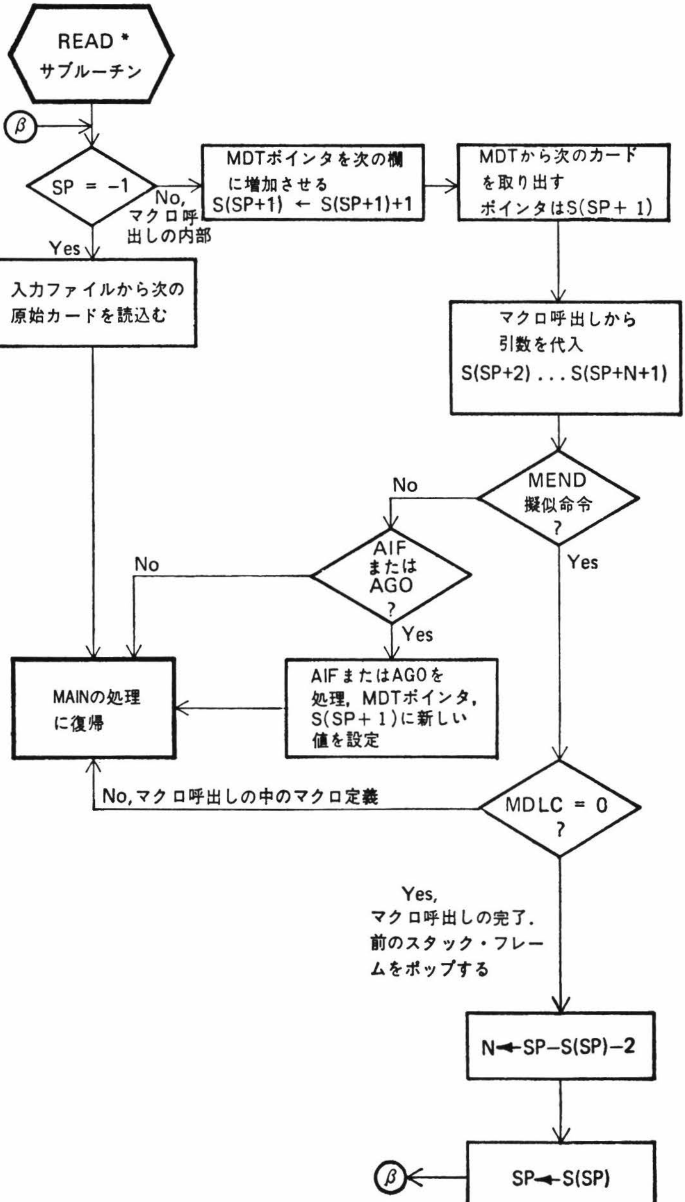


図 4.8　回帰的なマクロ展開のための READ 機能の詳細

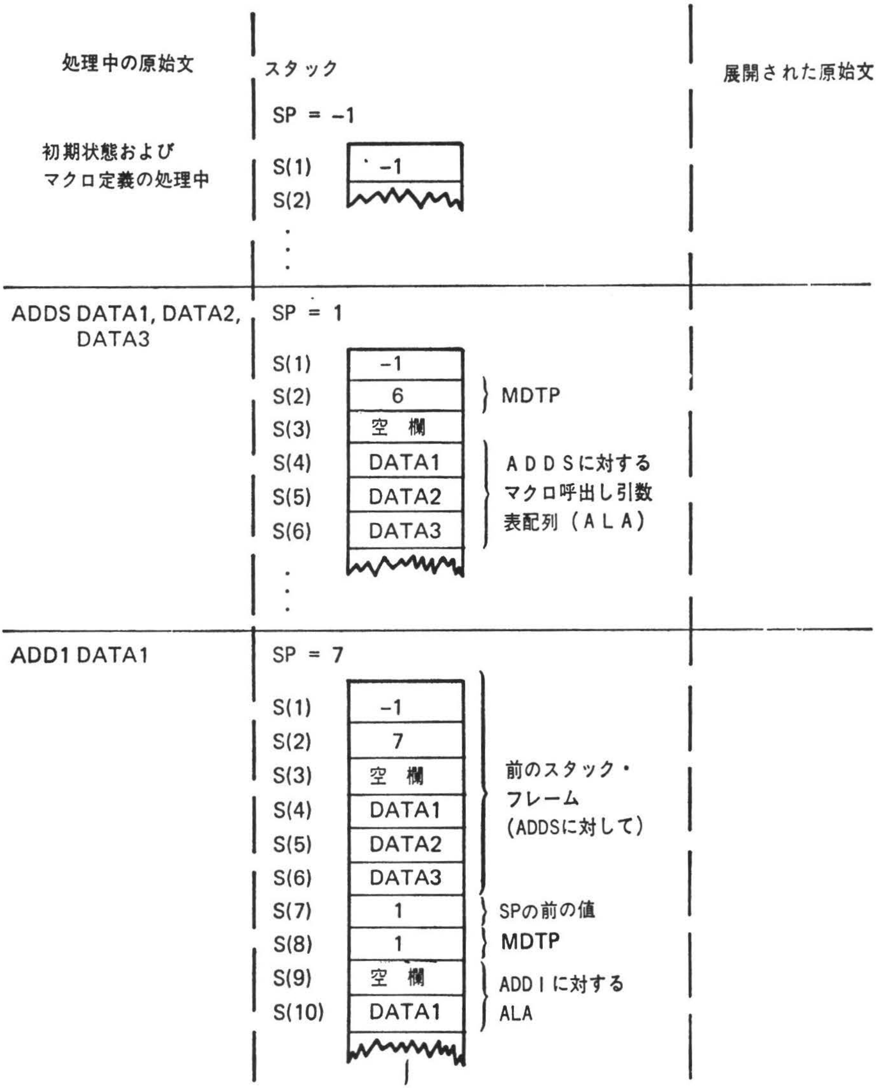


図 4.9(a) 例5の処理の進行（第1部）

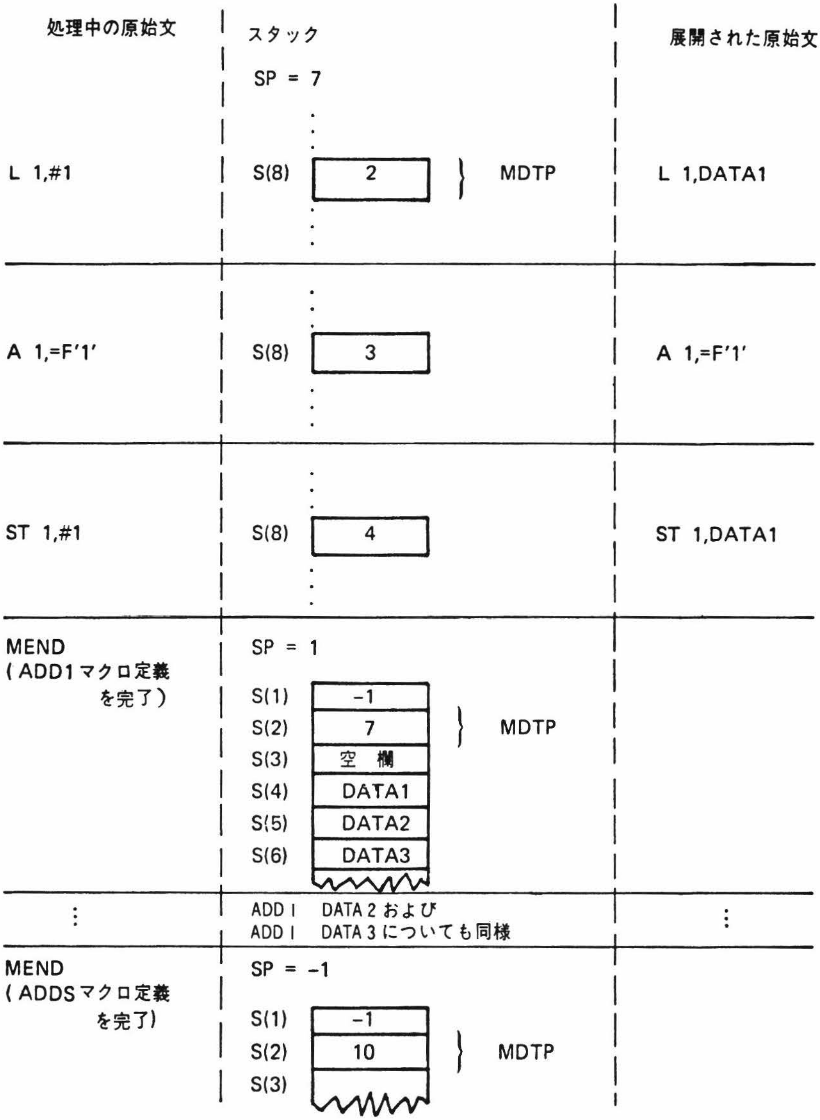


図 4.9(b)　例5の処理の進行（第2部）

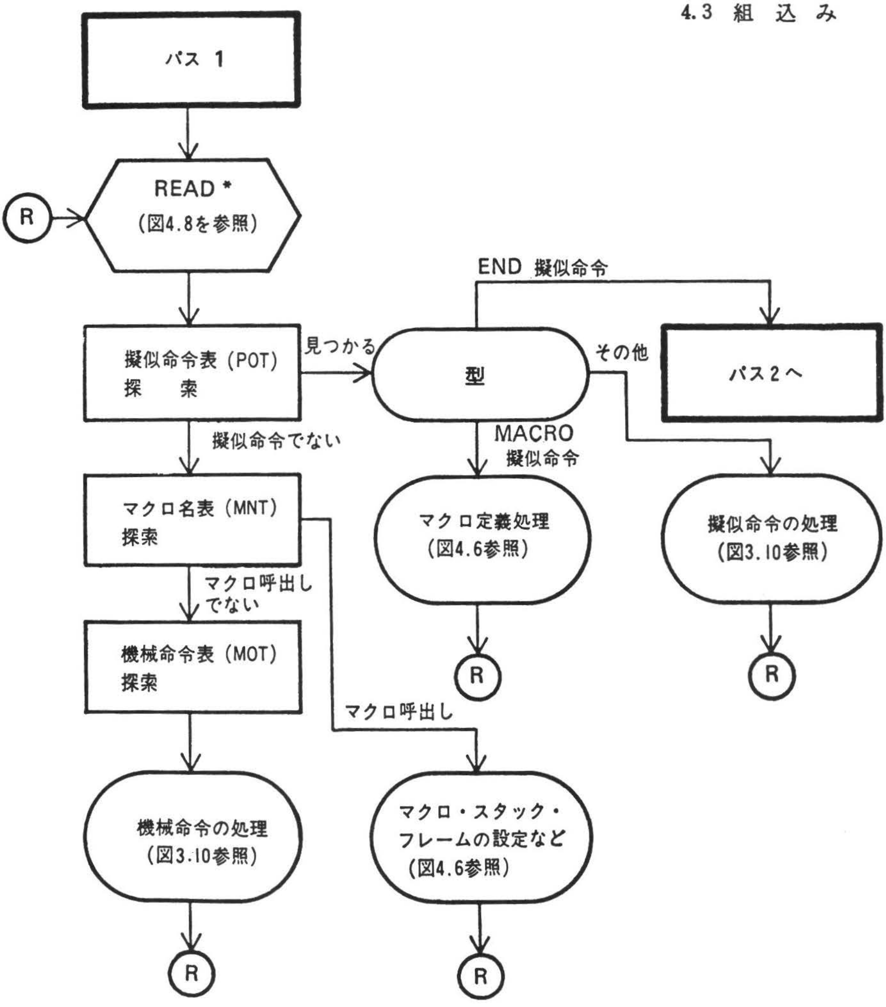


図 4.10 アセンブラのパス1と組合せたマクロ・プロセッサ

また，読者はマクロ原始プログラムを展開するアルゴリズムを用いてマクロ・プロセッサの動作を模擬してみるのが望ましい．**図 4.9** は例5の処理の種々の時点でのスタックの状態と，展開された原始プログラムを示している．この例に対応するマクロ定義表（MDT）は **図 4.5** に示されている． 読者はマクロ展開のすべてのステップをたどり，**図 4.9 a** および **図 4.9 b** の結果と比較して

みてほしい．

4.3.4　アセンブラの中への組込み

マクロ・プロセッサは，アセンブラのパス1の前に入力テキストを完全に調べるプリ・プロセッサとして，アセンブラに加えることができる．また，マクロ・プロセッサは，アセンブラのパス1の中に組込むこともできる．

マクロ・プロセッサをパス1の中に組込むと，中間ファイルのオーバヘッドを除くことができ，類似した機能を結合してマクロ・プロセッサとアセンブラの複合体を改良することができる．例えば，マクロ名表（MNT）はアセンブラの命令コード表（MOT または POT）と結合でき，各欄のフラグでマクロ名であるか否かの表示を行なうことができる．マクロ・プロセッサへの MACRO 擬似命令はアセンブラの正規の擬似命令処理ルーチンで検出可能である．マクロ呼出しを展開し，原始入力を受取る入力の READ の機能は**図4.8**のものと同じである．**図4.10**はこのアルゴリズムの流れ図である．これを第3章の流れ図と比較せよ．

マクロ・プロセッサをパス1に含めることの主な利点は：

1. 多くの機能は2回組込む必要がない（例えば，カードの読込み，文の型の判定）．
2. 処理のオーバヘッドが小さい．機能が結合されているのでマクロ・プロセッサからの出力を保存してアセンブラへの入力とするための中間ファイルが不要である．
3. アセンブラの機能（例えば EQU 文）をマクロと関連させて使用することができるので，もっと柔軟性に富むものとすることができる．

主な欠点は：

1. アセンブラのパス1とマクロ・プロセッサを結合すると，プログラムが大きすぎて機械によってはコアに入れることができない．
2. プログラムがあまりにも複雑になる．プログラマは2班に分れてアセンブラのパス1とマクロ・プロセッサを作成し能率を上げることができるが，2つの機能を結合したものは1つの班または1人のプログラマには

大きすぎで手に負えない．

アセンブラの中にマクロ機能を組込む検討を以上で終りにして，次にマクロ・プロセッサに新しいパスを追加することを考える．多くのマクロ・プロセッサは“プリ・パス”を持っている．その機能はデータの型などの原始プログラムのある種の性質を調べることである．このようなマクロ・プロセッサがあるとプログラマは条件付きマクロ擬似命令を使用することができる．この擬似命令はプログラムの性質に応じた条件によってマクロを展開する．

4.4 ま と め

マクロ言語とそれに対するプロセッサはプログラム言語の見掛けの形を変えてしまう．マクロ・プロセッサはアセンブラと共に使用するとき，プログラマに数多くの便利な道具を与え，自分専用の“高水準”言語の定義を可能にする．

マクロ・プロセッサの果さねばならない4つの基本的な役割りは，

1. マクロ定義の認識
2. マクロ定義の保存
3. マクロ呼出しの認識
4. マクロ呼出しの展開と引数の代入

アセンブラ言語に対するマクロ・プロセッサは種々の方法で組込むことができる．本章では3つの方法を示した．

1. 独立の2パス・プロセッサ
2. 独立の1パス・プロセッサ
3. 標準の2パス・アセンブラのパス1に結合したプロセッサ

演 習 問 題[1]

1. マクロ・プロセッサの入力と出力について述べよ．アセンブラの原始コー

---

1 * は IBM 360 マニュアルの使用を要する問題であることを示す．

ドの書式とどのように関係しているか.

2. 下のプログラムについて次に答えよ.

1) 展開したアセンブラ語のプログラム（すなわち，マクロを含まない形）

2) マクロ処理の後の MDT

3) 処理の後の MNT

a. b. で用いられている SETA 擬似命令は問題6に示されている.

```
         MACRO
         XYZ        &A
         ST         1,&A
         MEND
         MACRO
         MIT        &Z
         MACRO
         &Z         &W
         AR         4,&W
         XYZ        ALL
         MEND
         ST         &Z,ALL
         MEND
PROG     START
         USING      *,15
         MIT        HELLO
         ST         2,3
         HELLO      YALE
YALE     EQU        5
ALL      DC         F'3'
         END
```

b. 指数関数を計算する．例えば 5^3.

```
         MACRO
         EXPO       &EXP
         LCLA       &N
&N       SETA       &EXP
         AIF        (&N EQ 1) .STOP
         MR         0,2
&N       SETA       &N-1
         EXPO       &N
.STOP    ANOP
         MEND
EXPON    START      0
         USING      *,15
         L          2,BASE
         L          1,BASE
         SR         0,0
         EXPO       3
         ST         1,ANS
         BR         14
ANS      DS         F
BASE     DC         F'5'
         END
```

3. 本章で示したマクロ・アセンブラは2とおりのデータ・ベースを用いている．1つは何ら修飾をせずにデータを参照し，他はデータを作ったり修飾したりする．下の行列にはアセンブラで使用される種々のデータ・ベースと，それらの使用法が記されている．第4.3.4節のようにアセンブラのパス1とマクロ・プロセッサが結合していたとして対応する欄にレを付けて表を完成せよ．

	パス1で作成，変更しなければならない	パス1で参照されるが変更されない	パス1で作られパス2のために保存される	パス2で変更されない	パス2で作成，変更される	パス2で参照される
記号表						
リテラル表						
ベース・レジスタ表						
中間テープ						
MNTを含む機会命令表（MOT）						
疑似命令表						
入力デック						
マクロ定義表（MDT）						
ロケーション・カウンタ						
マクロ・スタック						

4. マクロ展開はある意味でプログラム実行中のサブルーチン呼出しときわめてよく似ている．類似点と相違点を述べよ．サブルーチン呼出しのときのスタック・フレームに相当するものは何か．

5. a. アセンブラがマクロ呼出しを処理しているとし，このマクロはマクロ定義を含んでいるとする．新しいスタック・フレームが作成されるか．作られるとするとどのような情報が格納されるのか．

b. このように入れ子の定義を処理するとき，他のデータ・ベースは変更されるか．されるとすればどれがどのようにか．

c. 入れ子の定義に対するMENDカードを処理した後に，外側のマクロでこのマクロの呼出しが可能か．

d.　そのマクロは外側のマクロの任意の場所で使用可能か．そのマクロはプログラムの任意の場所で使用可能か．

e.　マクロの中でマクロ定義を認めないとしたら，アセンブラからマクロ・スタックを取り除くことができるか．理由と共に答えよ．

f.　マクロのどの特徴がスタックと回帰性を必要とするのか．

6.　マクロ定義の中にラベルを認めるとき，本章で示されたマクロ・プロセッサの組込みをどのように修正すればよいのか．

7.　a.　1パスでマクロ処理が可能とするために，マクロ言語に制限を設けなければならない．マクロに対する制限事項と，プログラム構成に与える制約を述べよ．

b.　1パス・マクロ・プロセッサは条件付きマクロ擬似命令によって呼出されたマクロの処理を行なえるか．

次の場合を考えてみよ：

```
        MACRO
        WCM     &S
          :
        AIF     (&S EQ 19)  .END
          :
.END    MEND
          :
        WCM     V
          :
V       EQU     12
          :
        END
```

c.　1パス・プロセッサで不可能ならば，どのような変更をすればこの場合を取り扱えるようになるか．

8.　a.　マクロ・プロセッサのみを使用するアセンブラを組込みたい（すなわち，マクロ・プロセッサはアセンブラ語を受理して 16 進機械コード

を出力する)．ただの２つの擬似命令 (SET，BYTE)，および，２つのマクロ擬似命令 MACRO および MEND を持つマクロ・プロセッサを考えよ．

２つの擬似命令は：

(１) α SET β ——ラベル欄の記号αは式βの値を割当てられる．記号の値は他の SET によって設定し直すことが可能である．記号 LC はアセンブラのロケーション・カウンタを示す．

(２) BYTE **式(基数,容量),〔式(基数,容量)〕**——BYTE 擬似命令は８ビットの機械バイトをロケーション・カウンタに対応した場所 (@LC) に生成する．演算数欄の左から右への順で示されるサブフィールドによりバイトは構成される．**基数**は式が16進か10進かを指定する．**容量**はサブフィールドの長さが（８ビットの内の）何ビットであるかを示す．

例：AR 命令（レジスタにレジスタの内容を加算する)． 命令の16進コードを 'IA' とする．

```
          MACRO
&VAR1     AR        &VAR2, &VAR3
&VAR1     SET       LC
          BYTE      1A (16,8)
LC        SET       LC+1
          BYTE      &VAR2 (10,4), &VAR3 (10,4)
LC        SET       LC+1
          MEND
```

'AR' が MACRO として定義された後にこの“アセンブラ”は AR 命令を処理することができる（すなわち，AR 命令を受理し，等価な16 進機械語を生成する)．

SR（レジスタ減算）を定義せよ．

b. IBM 360 の命令A，L（加算，ロード——第２章参照）を定義せよ．

9. マクロを構成するとき，その文字列を保存しておけば後で使用することができ便利である．このような機能を本章で論じたようなアセンブラによって実現するために，次のものを追加したい．

1) 原形文の '仮の'，または，'変数' の記号（文字“&”で始まるもの）に

加えて，2つの文，GLOBAL 文と LOCAL 文，で変数記号を作ることができること．これらの文の構文は，

GBLA　　　&記号
　　　または
LCLA　　　&記号

ただし，“&記号”は“設定記号”と呼ばれる変数記号である．

GBLA (GloBaL　Arithmetic) 記号は，同じ名前を参照するすべてのマクロにおいて，その記号には同じ値が与えられるという意味で，“全体的”な値を持つことを宣言する．この“設定記号”は単にアセンブラのマクロ処理段階でのみ値を持つだけであるが，実行プログラムの中でラベルをEXTRN と宣言することに非常によく似ている．

LCLA (LoCaL　Arithmetic) 記号は，それを含むマクロが呼ばれる度に0に初期化され，同じ名前の“設定記号”を持つ他のマクロは別の値を持つことを宣言する．

算術記号は32ビット語の2進整数として格納されて処理される．

2)　次によって“設定記号”に値を設定する．

&設定記号　　SETA　　算術式

この文は式の値を“設定記号”に割当てる．

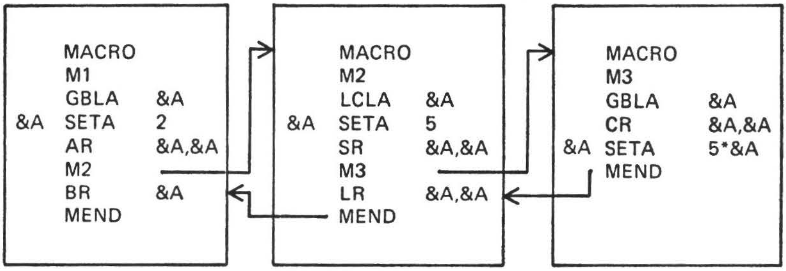
LCLA，GBLA および SETA 擬似命令が，次頁の図の M1 の呼出しでどのように働くかをよく理解してから次の問に答えよ．

次の問　a，b，c，d　に対して，マクロ・プロセッサはアセンブラのパス1の中に組込まれ，第4.3.3節に論じられているのと同様のスタックを持っていると仮定してもよい．

a.　1)　各擬似命令に対してアセンブラのどのパスを変更しなければならないか．

　　2)　各擬似命令に対してマクロ処理のどの段階（定義または展開）を変更しなければならないか．

b.　1)　局所記号（LCLA）は，多くのマクロ・パラメータと同様にスタ




```
.
.
.
25      M1
26+     AR     2,2
27+     SR     5,5
28+     CR     2,2
29+     LR     5,5
30+     BR     10
```

（＋は原始コードになく，生成された文を示す）

ック・フレームに欄を追加するだけで組込むことができるか．できないとすればどうすればよいか．読者の方法と，その方式を選んだ理由を説明せよ．

2） b. 1)の問を汎用記号（GBLA）に対して答えよ．

c. SETA 命令を処理するための方法を述べよ．局所および汎用記号の両方を考慮し，どのようにして正しい変数が処理されるのかを考えよ．

d. 1） LCLA 文はどのように処理すればよいかを述べよ．

2） GBLA 文についても同様のことを述べよ．

10. *本章で示されたマクロ・プロセッサ（すなわち，1パス，2パス，アッセンブラとの結合による組込みの各々に対して）の設計を変更して AIF および AGO 文が取り扱えるようにせよ．

第 5 章

ロ　ー　ダ

本章の目的は各種のローダの方式を論じ，最も一般的な直接結合型ローダの設計を示すことである．

前章で説明したように，利用者の**原始プログラム・**デックは通常アセンブラやコンパイラによって**目的プログラム・**デック（機械語）に変換される．**ローダ**は目的プログラム・デックを受理し，これらのプログラムを計算機で実行するための準備を行ない，実行を開始させる（**図 5.1** 参照）．

特に，ローダは次の 4 つの機能を果さなければならない：

1. プログラムのために記憶装置の領域を割付ける（**割付け**：allocation）．
2. 目的デックの間の記号参照を解決する（**結合**：linking）．
3. 番地定数などの番地に関係する場所を，割付けられた領域に適合するように調整する（**再置配**：relocation）．

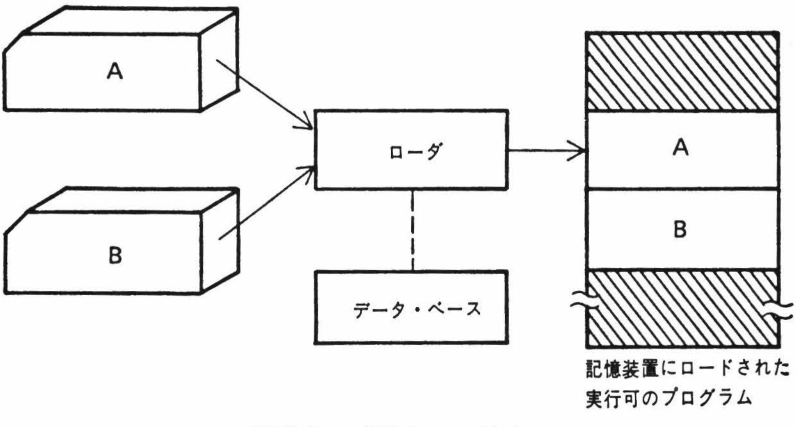


図 5.1　一般的なロード方式

4. 物理的に機械命令とデータを記憶装置の中へ置く（**ローディング**）.

本節で使用される大半の例は IBM 370 のアセンブラおよびローダに基づいている. 他の幾つかのローダ方式は, IBM 7094, IBM 1130, UNIVAC 1108 および GE 635 などの固定語長, 直接番地指定形式の命令を持った計算機に基づいて論じられている.

簡単のために入力はカード・デックとしているが, もちろん実際にはテープやその他の2次記憶装置上のカード・イメージでもよい.

5.1 ローダの方式

本節では, ローダの4つの機能を達成するための各種の方式について論じる. **セグメント**という語をよく用いるが, これは1かたまりのものとして取り扱われる情報の単位をいい, その内容はプログラムであってもデータであってもよい. 通常, セグメントは1つの原始または目的デックに対応している. しかし, アセンブラの CSECT（セクション制御）擬似命令, FORTRAN の COMMON 文, または PL/I の EXTERNAL STATIC データ属性を用いて, 複数のプログラムやデータ・セグメントを1つの原始デックの形で作成することができる.

5.1.1 "コンパイル・実行" 型ローダ

ローダ機能を実行する1つの方法は, 記憶装置の1部でアセンブラを実行し, アセンブルされた機械語とデータをアセンブル時に直接割付けた番地に置く方法である（**図 5.2**）. アセンブルを完了したときに, アセンブラはプログラムの開始命令に制御を渡す. これは余分の手続きを必要とせず簡単な解決法である. この方法は WATFOR FORTRAN や他の幾つかの言語プロセッサにおいて用いられている.

このようなローディング方式は一般に "コンパイル・実行" 型または "アセンブル・実行" 型と呼ばれ, 比較的組込むのが容易である. アセンブラは単にコードをコアの中に置き, "ローダ" は新しくアセンブルされたプログラムへ制御を渡す1つの命令から成る.

しかしながら, これには明らかに幾つかの欠点が存在する. 第1, アセンブ

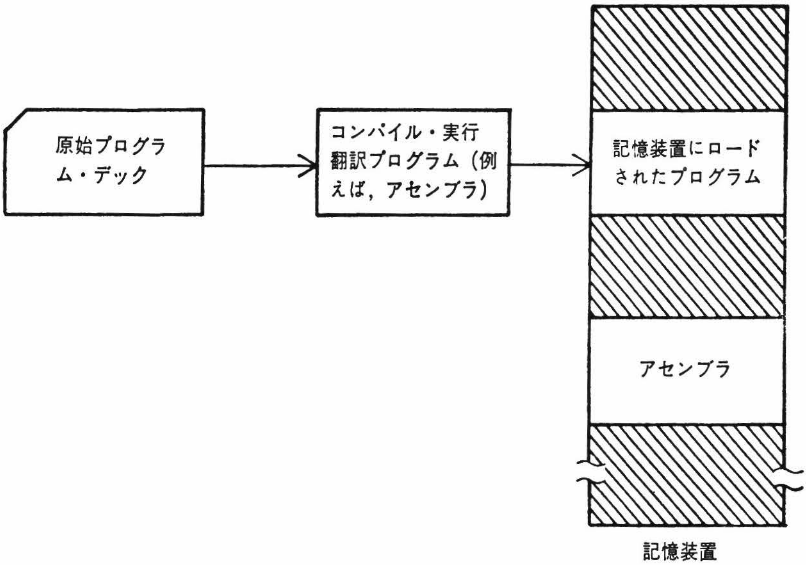


図 5.2 "コンパイル・実行" 型ローダの方式

ラによって占有されているコアが目的プログラムで使用できず，記憶装置の一部が無駄使いされている．第2，利用者のプログラム・デックを実行の度に翻訳（アセンブル）しなければならない．第3，複数セグメントを処理することが，特に，原始プログラムが違った言語（例えば，1つのサブルーチンはアセンブラ語で，他は FORTRAN や PL/I など）で書かれているときはきわめて困難である．最後の欠点はアセンブラの設計で論じたモジュラー・プログラムの作成を非常に困難とする．

5.1.2 一般的なローダの方式

アセンブルされた命令やデータをいったん外部へ出力すると，アセンブラがコアを浪費する問題を解決することができる．この出力カードを保存しておけば，このプログラムを実行したいときにいつでもロードすることが可能である．アセンブルされたプログラムは（翻訳を完了しているので）アセンブラが使用していたのと同じコアの領域にロードできる．符号化された命令を記録し

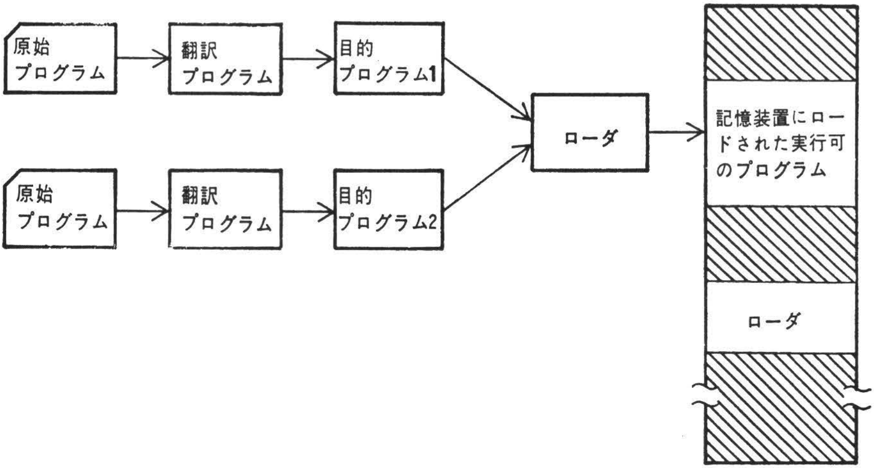


図 5.3　一般的なローダ方式

たカードなどの出力を**目的デック**と呼ぶ.

前述の“コンパイル・実行”方式の欠点の1つを取り除くため，中間データとして目的デックを使用すると，システムにローダという新しいプログラムを追加せねばならない(図5.3). **ローダ**はアセンブルされた目的プログラム中の機械命令，データ，およびその他の情報を受理し，計算機で実行できる形としてコアの中に機械命令とデータを置く．ローダはアセンブラより小さいと考えられるので，利用者にはより大きい記憶装置が使用できる．さらに，後日プログラムを実行するために，アセンブルのやり直しをしなくてもよいという利点がある.

最後に，すべての原始プログラム翻訳ルーチン（アセンブラやコンパイラ）が同じ形式の目的プログラム・デックを生成し，同じ形式の結合規約を使用すれば，ローダによって処理される目的デックはすべて同じ“言語”（機械語）で書れていることになるので，原始プログラムの段階ではサブルーチンを幾つかの違った言語で作ることが可能となる.

5.1.3 絶対型ローダ

図5.3の一般的なモデルに適合する最も簡単なローダ方式は絶対型ローダと呼ばれるものである．この方式では，アセンブラは原始プログラムの機械語への翻訳結果を“アセンブル・実行”法とほとんど同じ形で出力する．異なっている点は，直接記憶装置に置く代りにデータはカードにせん孔される（目的デック）ことである．これに対してローダは単に機械語のテキストを受取り，アセンブラによって定められたコアの場所に入れる．この方式では，アセンブラはロード時に記憶装置の中にないので，利用者がより大きいコア領域を使用できる．

絶対型ローダは組込みが簡単であるが幾つかの欠点を持っている．第1，プログラマはプログラムがロードされるべきコアの番地をアセンブラに指示しなければならない．その上に，複数のサブルーチンがあればその番地を記憶し，サブルーチン結合を行なうためにその番地を陽に指定して使用しなければならない．

図5.4は絶対型アセンブラと絶対型ローダの動作を説明している．プログラマは，2つのサブルーチンに同じ番地を重複して割付けないよう注意しなければならない．

プログラム MAIN は番地100〜247を，サブルーチン SQRT は番地400〜477を割付けられている．MAIN が変更されて長さが300バイト増加したとすると，MAIN の終り（100＋300＝400）は SQRT の始まり（400）に重なる．そこで START 擬似命令カードを変更してアセンブルし直し，SQRT に新しい場所を割付けなければならない．その上，SQRT の番地を参照している他のすべてのサブルーチンを変更することも必要である．数多くのサブルーチンが使用されている場合には，このように手で“やりくり”をすることはきわめて複雑で，退屈で，しかもコアの無駄使いとなることがある．

絶体型ローダではローダの4つの機能は次のようにして実行される：

1. 割付け——プログラマによる．
2. 結合——プログラマによる．
3. 再配置——アセンブラによる．

MAINプログラム

```
MAIN     START    100
         BALR     12,0
         USING    MAIN+2,12
           :
         L        15,ASQRT }  SQRT
         BALR     14,15    }  呼出し
           :
ASQRT    DC       F'400'   }  SQRTの
         END               }  番　地
```

SQRTサブルーチン

```
SQRT     START    400
         USING    *,15
           :               }  平方根を計算
         BR       14       }  復　帰
         END
```

絶対アセンブラへの原始デック入力

MAINプログラム

番地	命令	
100	BALR	12,0
102	⋮	
⋮		
120	L	15,142(0,12)
124	BALR	14,15
126	⋮	
⋮		
244	F'400'	
248		

SQRTサブルーチン

番地	命令	
400		
⋮	⋮	
476	BCR	15,14
478	⋮	

絶対アセンブラからの目的デック出力

(a)

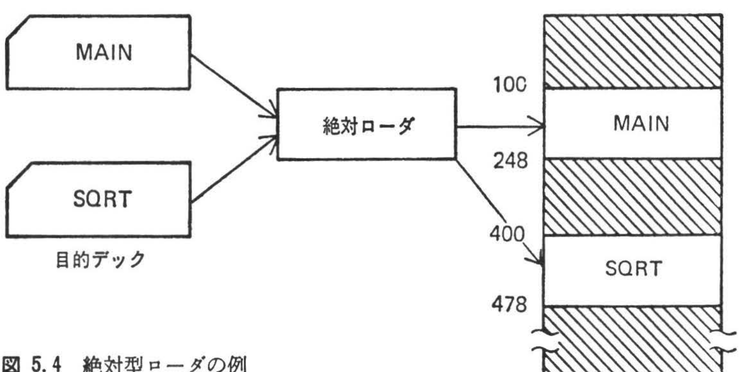


図 5.4　絶対型ローダの例

(b)

4. ローディング——ローダによる.

5.1.4 サブルーチンの結合

本節では，プログラマの観点から，アセンブラ語のプログラムにおけるサブルーチン呼出しの特別な機構を簡単に論じる.

サブルーチン結合の問題とは次のようなものである．プログラムAはサブプログラムBに制御を渡したい．プログラマは，プログラムAの中でサブプログラムへの移行命令(例えば BAL 14,B) を書くことができる．しかし，アセンブラは記号名参照に対してその値がわからず，特別の機構が用意されていないときには誤り（未定義記号）として宣言する.

再配置型や直接型ローダにはこの機構が組込まれている．アセンブラの擬似命令 EXTRN はそれに引続く記号がこのプログラムの中で参照されるが，定義は他のプログラムで行なわれていることを示す．これに対応して，あるプログラムで記号が定義され，他で参照されるとき，この記号を擬似命令 ENTRY に続けて宣言する．そして，アセンブラはローダに対して，これらの記号は他のプログラムによって参照されることを通知する．例えば，次の一連の命令は他のプログラムを呼出すための簡単な呼出し手順の1つである.

```
MAIN     START
         EXTRN    SUBRQUT
         ---
         ---
         L        15,=A(SUBROUT) }
         BALR     14,15          }   CALL SUBROUT
           .
           .
           .
         END
```

上述の一連の命令で SUBROUT は外部変数である．すなわち，SUBROUT はこのプログラムの中で参照されるが，ここでは定義されていないということが最初に宣言されている．ロード命令は変数の番地をレジスタ 15 にロードする．BALR 命令はレジスタ 15 の内容にしたがって SUBROUT の番地に分岐し，レジスタ 14 に次の命令の番地を残す．しかし，たいていのアセンブラにおいては単に CALL SUBROUT というようにマクロを使用することができる．このマクロはアセンブラによって上に示したような呼出し手順に翻訳される.

このように呼出す方と呼出される方のプログラムが共に協同して働くことが必要であるので，プログラミングの規約を設けておかなければならない．IBM 360 の規約は，結合のためおよびベース・レジスタとしてレジスタ 15 を使用している．上のプログラムでは，呼出す方がレジスタ 15 の中に呼出される方の開始番地をロードしていることに注意してほしい．したがって，呼出されたサブルーチンはベース・レジスタをロードする必要はない． レジスタ 14 は呼出したプログラムへの復帰番地を納めている．サブルーチンの中でレジスタ14を他の用途に使用するときには，内容を退避した上で使用し，復帰の前に必らず復元しておかなければならない．典型的なサブルーチンの手順は次のとおりである：

```
SUBROUT    START
           USING    *,15
             :
           BR       14
           END
```

プログラムの開始番地が既にレジスタ 15 にロードされているので BALR 命令は不要である．BR 14命令は，呼出しプログラムの復帰番地としてレジスタ14に納められている番地への無条件分岐である．付録において IBM 360 で使用されている方法の詳細を示し，呼出し手順について議論を進めている．次に述べるのは基本的な機構である：

サブルーチンと入口の名前付け(STARTおよびENTRY擬似命令)

```
A      START     サブルーチンAの定義
       ENTRY     B1,B2,B3,... は番地 B1,..., Bnが
                         サブルーチンの他の入口で
                         あると定義する。
B1     -----
B2     -----
```

アセンブラの結合擬似命令

複数の入口を設けることは次の利点がある．

1. プログラムの共有を可能にする．

例： SIN および COS は基本的には同じ計算を行なうので同じルーチンの異なる入口を用いる方法が使用できる．

2. 関連したルーチンを1まとめにして使いやすくする．
3. 共通のデータ・ベースを参照するのに都合が良い．

サブルーチンの参照（EXTRN 擬似命令）

アセンブラの記号には内部記号と外部記号とがある．外部記号はアセンブラにはその値がわからず，ローダによって与えられる．ローダの動作は次節で述べる．

EXTRN　E1, E2, ……

は E1, E2, などが番地定数として使用される外部記号であると定義している．

例:　CALL BETAは次のように展開される

```
      EXTRN   BETA
        :
      L       15,ABETA
      BALR    14,15
        :
ABETA DC      A(BETA)
```

5.1.5 再配置型ローダ

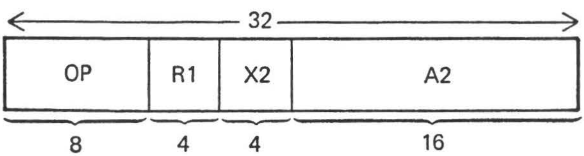
1つのサブルーチンが変更されたとき，すべてのサブルーチンのアセンブルをし直さなくてもよいようにし，プログラマに代って割付けと結合の役割りを果すために**再配置型ローダ**が開発された．再配置型ローダ方式の例としてはIBM 7094，IBM 1130，GE 635 および UNIVAC 1108 などで用いられたBSS（Binary Symbolic Subroutine：2進記号サブルーチン）ローダがある．BSSローダでは，手続きセグメントは複数個使用できるが，データ・セグメント（共通セグメント）はただ1つだけしか使用できない．アセンブラは各手続きセグメントを独立にアセンブルし，テキストと，再配置およびセグメント間参照に関する情報とをローダに渡す．以下では典型的なBSSローダについて述べる．

BSS方式を使用する再配置型アセンブラの出力は，目的プログラムと，他のプログラムを参照するために必要なすべての情報，および，任意の場所に置かれたとき変更を要する部分に関する情報，すなわち，コアの割付けに関係する再配置情報を持っている．

各原始プログラムに対してアセンブラは，プログラムを機械語に翻訳したテキストを出力する．このとき，原始プログラムで参照されたサブルーチンの名前を納めている語の配列をテキストの先頭に付けて出力する．この配列のことを**トランスファ・ベクトル**という．例えば平方根ルーチン(SQRT)が参照されたとし，これが最初に呼出されたサブルーチンであるとすると，トランスファ・ベクトルの最初の場所は記号名 SQRT を納めている．SQRT を呼出す文は，SQRT に関するトランスファ・ベクトルの場所へ分岐する命令に翻訳される．

また，アセンブラはローダにプログラムの長さやトランスファ・ベクトルの長さなどの付加的情報をも用意する．テキストとトランスファ・ベクトルをコアにロードした後で，ローダはトランスファ・ベクトルに示されている各サブルーチンをロードする．次にトランスファ・ベクトルの各欄に，対応したサブルーチンへの制御移行命令を置く．したがって，CALL SQRT 文を実行すると，まず，トランスファ・ベクトルの最初の番地に分岐し，次に SQRT の入口へ分岐する．

BSSローダ方式は，しばしば固定長の直接番地指定命令形式の計算機で用いられている．例えば，IBM 360 RX 命令の形式が次のようであると仮定する：



ただし，A2 は16ビットで演算数の絶対番地を指定する． これは**直接番地指定命令**形式である．これは，多くの初期の計算機，今日の“ミニ・コンピュータ”や“ミディ・コンピュータ”などの場合のように記憶装置の容量が $2^{16}=65,536$ バイト以下ならば正しく働く．

再配置を行なうためにはすべての命令の番地部を処理する必要があるので，直接番地指定命令形式の計算機では IBM 360 よりも再配置が困難である．

IBM 360 型のベース・レジスタがないときには，この問題は **“再配置ビット”** を使用して解決されている．アセンブラはこのビットを各命令または各番地部に対応させ，このビットが1であれば，番地部の再配置を行ない，0であれば再配置をしないことにする．この再配置表示子は再配置ビットと呼ばれ，目的デックに含められている．

図 5.5 には BSS ローダを使用する仮想の“直接番地指定型”360 のために

プログラムの長さ = 48 バイト
トランスファ・ベクトル = 8 バイト

原始プログラム				相対番地	再配置	目的コード	
MAIN	START						
	EXTRN	SQRT		0	00	'SQRT'	
	EXTRN	ERR		4	00	'ERR*b*'	
	ST	14,SAVE	復帰番地を保存	8	01	ST	14,36
	L	1,=F'9'	試験データをロード	12	01	L	1,40
	BAL	14,SQRT	SQRTを呼出す	16	01	BAL	14,0
	C	1,=F'3'	答えを比較	20	01	C	1,44
	BNE	ERR	ERRに制御を移す	24	01	BC	7,4
	L	14,SAVE	復帰番地を求める	28	01	L	14,36
	BR	14	呼出し者に復帰	32	0	BCR	15,14
				34	0	(整合のために飛ばされた)	
SAVE	DS	F	一時記憶	36	00	(一時記憶)	
	END			40	00	9	
				44	00	3	

図 5.5 “直接番地指定型”360に対するプログラムのアセンブル

書かれた簡単なアセンブラ語のプログラムが示されている．プログラムの機能は重要ではない．プログラムは SQRT サブルーチンを呼出し，9の平方根を求める．結果が3でなければサブルーチン ERR を呼出す．この計算機は直接番地指定型であるから目的コードにベース・レジスタ部はなく，原始プログラムの中で USING 擬似命令を使用する必要はない． EXTRN 擬似命令は SQRT と ERR が他のサブルーチンの名前であると定義する．これらの記号の番地は，

このサブルーチンの中では定義されていないので，**外部記号**と呼ばれる．アセンブラは各外部記号に対して4バイトの語をプログラムの先頭に生成し，記号に対する EBCDIC 文字を入れる（簡単のために，記号は4文字よりは長くはないと仮定する）．この語はトランスファ・ベクトルと呼ばれる．外部記号を参照する命令の番地部には対応したトランスファ・ベクトル語の番地が割付けられる．またアセンブラは，プログラムの中のすべての半語（2バイト）に対しそれぞれ1ビットの再配置ビットを生成する．例えば，アセンブルされた命令

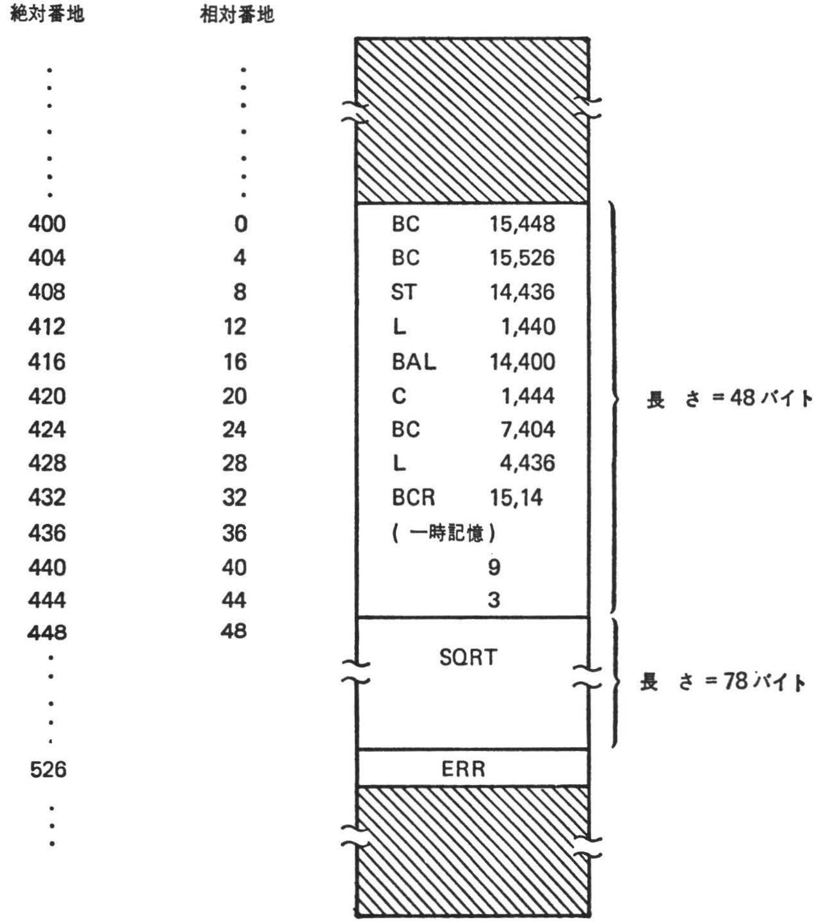


図 5.6 "直接番地指定型" 360に対するプログラムの BSS ローディング

ST 14,36 では，第1の半語は命令コード，レジスタおよびインデックス部を持ち再配置されないが，第2の半語は相対番地36を持ち再配置が必要である．したがって，再配置ビットとして01が割付けられている．

図5.6 は，BSS ローダによってプログラムがロードされた後の記憶装置の内容を示している．ローダは再配置ビットに基づいて，400 番地に割付けられた MAIN の番地に関し，番地部を再配置している．ローダはプログラムの長さの情報を使用して次に空いている場所を求め，SQRT および ERR サブルーチンをそれぞれ番地 448 と 526 に置いている．最後に，トランスファ・ベクトル語は対応したサブルーチンへの分岐命令に変更される．このようにローダの4つの機能（割付け，結合，再配置およびローディング）は BSS ローダによって自動的に行なわれる．

再配置ビットは再配置の問題を，トランスファ・ベクトルは結合の問題を，そしてプログラムの長さは割付けの問題をそれぞれ解決するために使用されている．

BSS ローダ方式には幾つかの欠点がある．第1，トランスファ・ベクトルは制御の移行に対してのみ有効であり，外部データ（他の手続きセグメント内のデータなど）のロードや格納のためには適当でない．第2，トランスファ・ベクトルは記憶装置の中の目的プログラムの大きさを増大させる．最後に，前述したように BSS ローダは手続きセグメントの処理は行なうが，共有すべきデータ・セグメント参照の問題は解決しない．これについては，多くの BSS ローダで，しばしば **COMMON** と呼ばれる共通データ・セグメントを用いて可能としている．この機能は，通常，再配置ビットを半語毎に2ビット使用するように拡張して組込まれている．すなわち，ビットが01のときその半語は手続きセグメントに相対的に再配置され，10であれば共通のデータ・セグメントの番地に相対的に再配置される．ビットが00または11のときその半語は再配置されない．

5.1.6 直接結合型ローダ

直接結合型ローダは一般的な再配置型ローダで，今日最も広く使用されているローダの方式である．後のコンパイラの議論でシステムはこの種のローダを

使用すると仮定している．

直接結合型ローダを用いると，プログラマは複数の手続きセグメントと複数のデータ・セグメントを使用して，他のセグメントの中の命令やデータを全く自由に参照できる．また，柔軟なセグメント間参照が可能となり，個々のプログラムの翻訳が独立にできるようになる．

直接結合型ローダの設計は本章の後部で述べられている．本節では IBM 370 型にならって，このローディング方式をとるアセンブラの出力の一般形を示す．出力の書式はある程度任意であるが，アセンブラがローダに対して用意しなければならない情報は定まっている．アセンブラ（翻訳ルーチン）はローダに対して，手続きまたはデータのセグメント毎に次の情報を用意しなければならない：

1. セグメントの長さ．
2. 他のセグメントによって参照されるセグメント中のすべての記号と，セグメント中におけるそれらの相対番地を示した表．
3. セグメント中では定義されていないがその中で参照されるすべての記号の表．
4. 番地定数がセグメントのどこに位置しているかに関する情報と，その値の処理の方法．
5. 原始プログラムの機械コードへの翻訳結果と割付けられた相対番地．

図 5.7 に直接結合型のローディング方式を使用する簡単な例を示す．左側の列の原始プログラムはアセンブラによって翻訳され，右側の列に示された目的コードが生成される．ここでも図の翻訳結果を示すのに記号名を使用している．

図 5.7 のカード14には DC（定数定義）擬似命令が使用されている．アセンブラは TABLE の番地を値とする定数を作り，これを POINTER というラベルの付けられた場所に置く．この時点ではプログラムがどこにロードされるかわからないので，アセンブラは最終的な TABLE の絶対番地を知らない．しかし，アセンブラは TABLE がプログラムの最初から 28 バイト目であることを知っている．アセンブラは POINTER の所へ 28 を入れ，場所 POINTER の内容はプログラムが絶対番地 0 以外へロードされたときは正しくないということをローダに通知する．例えば，もしプログラムが番地 2000 にロードされた

とすると，ローダは POINTER の内容を 2028 に変更しなければならない．

図5.7のカード17には別の DC 擬似命令が使用されている．アセンブラはサブルーチン SUM の番地を値とする定数を作り，ASUM というラベルの付けられた場所に置く．しかし，アセンブラは，手続き SUM がどこに置かれるかわからないので，この定数を生成することができない．したがって，アセンブラはローダに対して，プログラムがロードされるときに SUM の最終的な絶対番地を指定された場所 ASUM に入れるように指定した情報を用意しなければならない．

プログラムには JOHN という名が付けられている．この JOHN は外部から参照される．すなわち，他のプログラムによって“呼出される”ことのある記

プログラム				翻訳結果		
カード番号				相対番地		
1.	JOHN	START				
2.		ENTRY	RESULT			
3.		EXTRN	SUM			
4.		BALR	12,0	0	BALR	12,0
5.		USING	*,12			
6.		ST	14,SAVE	2	ST	14,54(0,12)
7.		L	1,POINTER	6	L	1,46(0,12)
8.		L	15,ASUM	10	L	15,58(0,12)
9.		BALR	14,15	14	BALR	14,15
10.		ST	1,RESULT	16	ST	1,50(0,12)
11.		L	14,SAVE	20	L	14,54(0,12)
12.		BR	14	24	BCR	15,14
				26	——	
13.	TABLE	DC	F'1,7,9,10,3'	28	1	
				32	7	
				36	9	
				40	10	
				44	3	
14.	POINTER	DC	A(TABLE)	48	28	
15.	RESULT	DS	F	52	——	
16.	SAVE	DS	F	56	——	
17.	ASUM	DC	A(SUM)	60	?	
18.		END		64		

図 5.7　アセンブラ原始プログラムとその翻訳結果

号である．また，記号 RESULT も他のプログラムから参照することができると宣言されている．

ここで述べる直接結合型のローディング方式の設計は IBM 370 の標準方式に類似している．アセンブラは目的デックの中にESD, TXT, RLDおよびENDの4種類のカードを生成する．外部記号辞書 (External Symbol Dictionary : ESD) カードは，このプログラムの中で定義され他の場所で参照できるすべての記号と，他で定義されこのプログラムの中で参照されているすべての記号とに関する情報を納めている．テキスト (TXT) カードは原始プログラムの翻訳版である実際の目的コードを納めている．再配置・結合ディレクトリー(Relocation and Linkage Directory : RLD) カードは，プログラムが置かれる番地に応じて変更を要する場所とその処理の仕方を示す情報を納めている．このような場所に対してアセンブラは，ローダが正しくその値を修正できるような情報を用意しなければならない．END カードは目的デックの終りを示し，アセンブルされたルーチンが主プログラムであるとき実行開始番地を指定する．図5.8は前のプログラムに対して ESD, TXT, RLD および END カードに現われる情報を示している．

図5.8の参照番号は実際にはカード上に現われない．これらは読者の便宜のために付けたもので，各々は目的カードを発生した原始プログラムのカード番号である．例えば，最初の RLD カードは原始プログラムのカード番号 14 によって生成されたものである．

図5.8に示されているように，プログラム JOHN には3枚の ESD カードが必要である．最初のカードはプログラムの名前 JOHN を持っており，この名前は外部から参照できる．“形式”記号は SD (Segment Definition : セグメント定義) であり，記号はセグメント定義名であることを示している．JOHN の相対番地は0で，プログラム JOHN の長さは64である．次の ESD カードには記号 RESULT が現われる．これは局所定義 (Local Definition : LD) であり，相対番地は52である．最後の ESD カードでは，記号 SUM は外部参照 (External Reference : ER) であることを示している． ER 記号が実際にどのように RLD カードと関連して使用されるかは後節で述べる．

TXT カードは実際にアセンブルされたプログラムを納めている．このカー

ESD カード

参照番号	記号	型	相対番地	長さ
1	JOHN	SD	0	64
2	RESULT	LD	52	—
3	SUM	ER	—	—

TXT カード

参照番号	相対番地	目的コード	
4	0	BALR	12,0
6	2	ST	14,54(0,12)
7	6	L	1,46(0,12)
8	10	L	15,58(0,12)
9	14	BALR	14,15
10	16	ST	1,50(0,12)
11	20	L	14,54(0,12)
12	24	BCR	15,14
13	28	1	
13	32	7	
13	36	9	
13	40	10	
13	44	3	
14	48	28	
17	60	0	

RLD カード

参照番号	記号	フラグ	長さ	相対番地
14	JOHN	+	4	48
17	SUM	+	4	60

目的デック

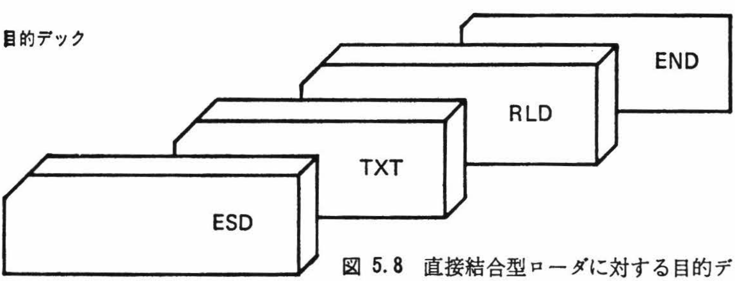


図 5.8　直接結合型ローダに対する目的デックの例

ドの書式と使用法は絶対型ローダの場合と似ている．

RLD カードは次の情報を含んでいる：

1. 再配置によって変更する必要のある定数の場所．
2. 何を用いて変更するか．
3. 変更処理．

例の最初の RLD カードは値 48 を納めており，変更が必要とされる定数の相対番地を示している．＋符号は定数に何かを加える必要があることを示し，記号欄は外部記号 JOHN に割付けられた値をこのセグメント内での相対番地 48 の内容に加える必要があることを示している．JOHN の相対値は 0 である．プログラムがロードされるとき，ローダはその絶対値を決定する．

2番目の RLD カードは値 60 を納めており，変更が必要とされる定数の相対番地を示している．記号欄は外部記号 SUM に割付けられた値を相対番地 60 の内容に加える必要があることを示している．アセンブラには SUM の絶対番地がわからないが，ローダは後で正しい値を入れることができる．

TABLE などの内部記号の番地定数を調整する処理は，通常，再配置と呼ばれる．これに対して，SUM のような外部記号に対して番地定数の内容を与える処理は，通常，結合といわれる．RLD カードの機構は両方の場合に使用されるので再配置・結合ディレクトリー・カードと呼ばれる．読者はこの技法と前に述べた BSS 再配置ローダで使用されている機構とを比較してみるとよい．

5.1.7 他のローダ方式――バインダ，リンク・ローダ，オーバレイ，ダイナミック・バインダ

前に示されたローダの方式には多数の変形がある．

直接結合型ローダの1つの欠点は，プログラムを実行するために毎回すべてのサブルーチンを割付け，再配置し，結合し，そしてロードする必要があることである．プログラムの中には多数のサブルーチンを含むことがあり，特に SQRT のようなユーティリティ・ルーチンを含むとき，このローディング方式は極端に時間のかかるものとなる可能性がある．さらに，ローダ・プログラムはアセンブラよりはるかに小さいとはいっても，依然としてかなりの領域を使

用する．これらの問題はローディングの処理をバインダとモジュール・ローダの2つの別のプログラムに分けることにより解決できる．

バインダはサブルーチンを“結合”することでは直接結合型ローダと同じ機能を果すが，再配置と結合を行なったテキストを記憶装置に直接置かずにファイルやカード・デックとして出力する．出力ファイルは直ちにロード可能な書式となっており，普通**ロード・モジュール**と呼ばれる．**モジュール・ローダ**は単にロード・モジュールを物理的にコアの中へ入れるだけでよい．バインダは割付け，再配置および結合の機能を果し，モジュール・ローダは単にローディングの機能を果す．

バインダには大別して2種類ある．最も簡単なものは単一の絶対型ローダのためのデックと非常によく似たロード・モジュールを生成する．プログラムのコア割付けはサブルーチンが結合されるときに行なわれてしまっている．この種のモジュールは実際のコアの1部分の“スナップ”または“イメージ”のように考えられるので**コア・イメージ・モジュール**と呼ばれ，対応するバインダは**コア・イメージ作成ルーチン**と呼ばれる．もっと複雑なバインダは**リンケージ・エディタ**と呼ばれ，再配置情報を記憶しておくことができるので再び出力されたロード・モジュールは，それを1つのものとして，再び再配置が可能であり，コアの任意の場所にロードすることができる．この場合モジュール・ローダはローディングのほかに付加的な割付けと再配置を行なわねばならないが，複雑な結合の問題にわずらわされることはない．

いずれの場合にも繰り返し使用されるプログラムは1回だけ結合を行なえばよく，必要なときにいつでもロードすることができる．コア・イメージ・バインダは比較的簡単で高速である．リンケージ・エディタは幾分複雑であるが柔軟な割付けおよびローディングが可能である．

ダイナミック・ローディング

これまで述べたローディングの方式では，必要なサブルーチンがすべて同時にコアにロードされるものと考えた．大きなプログラムのときや小さな計算機のときよく起こるように，もしこれら全部のサブルーチンのコア必要量が使用

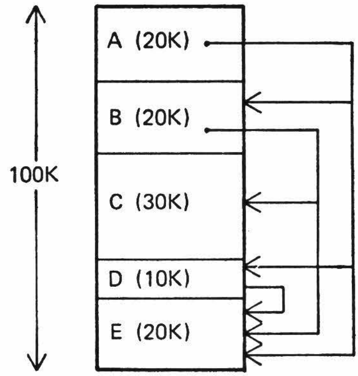
できる容量を上回れば困ったことになる．この問題を解決する試みとして，ページングやセグメンテーションなどの幾つかのハードウェア技法がある．これらは第9章で論ぜられる．本節では，ローディングの前にバインダを使用するダイナミック・ローディング方式を述べる．

1つのプログラムにおいて多くの場合幾つかのサブルーチンは別々の時刻に必要となる．例えば，アセンブラのパス1とパス2は互に排他的である．サブルーチンがどのサブルーチンを呼出すかを調べることによって，互に排他的なサブルーチンを明らかにする**オーバレイ構造**を作ることができる．**図5.9a** は5つのサブプログラム（A，B，C，D，E）から成り，コア必要量が100Kバイト必要なプログラムの例を示している．矢印はサブプログラムAがB，DおよびEだけを呼出し，サブプログラムBはCとEのみを，サブプログラムDはEのみを，それぞれ呼出し，サブプログラムCおよびEは他のルーチンを呼出さないことを示している．**図5.9b** では手続き間の独立性を強調している．注意すべきはBとDは決して同時には使用されないことである．同様なことはCとEについてもいえる．ある時点で実際に必要とされる手続きのみをロードすることとすれば，コアの所要量はオーバレイ構造の最も長い経路の長さに等しく，**図5.9b** の例では手続きA，BおよびCで70Kとなる．**図5.9c** はオーバレイ構造に一致した各手続きに対する記憶装置の割付けを示している．

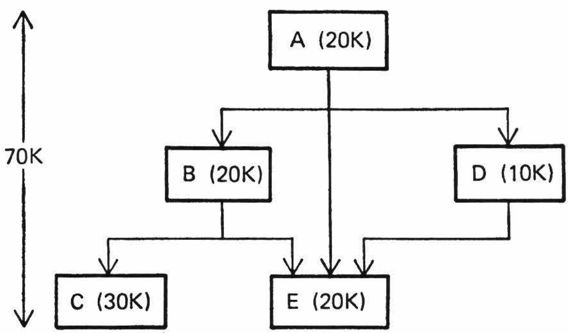
オーバレイ構造がうまく働くためには，モジュール・ローダは各種の手続きをそれらが必要になったときにロードしなければならない．ここではその詳細には立入らないことにするが，オーバレイ構造を処理し割付けを行なうことのできるバインダは多数存在することを付け加えておく．“呼出し”をいったん保留して必要な手続きをロードするローダの部分は**オーバレイ・スーパバイザ**または簡単に**フリッパ**と呼ばれる．また，この方式全体のことを**ダイナミック・ローディング**または**呼出し時ロード**（Load-on-call：LOCAL）という．

ダイナミック・リンキング

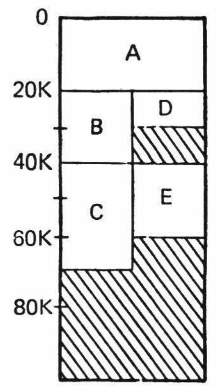
これまで述べたローディング方式の大きな欠点は，参照されたサブルーチンが実行されることがなくても（例えば，プログラムの中で呼出し文を入れたと



(a) 手続き間のサブルーチン呼出し



(b) オーバレイ構造



(c) 各手続きに対する記憶域の割当て

図 5.9 その他のロード方式

しても，条件が成立しなかったりして，この文が実行されないことがある)，ローダはこのサブルーチンを結合しなければならないことである．

その上にこれらの方式は，呼ばれるかも知れないすべてのプロシージャの名前をプログラマが陽に示さねばならないことである．次のようなプログラムを書くことは不可能である．

```
    ⋮
READ          SUBNAME, ARGUMENT
ANSWER =      SUBNAME(ARGUMENT)
PRINT         ANSWER
    ⋮
```

例でサブルーチンの名前（例えば SQRT, SINE など）は，他のデータと同様の入力パラメータ SUBNAME である．

第9章で論じるきわめて一般的なローディング方式はダイナミック・リンキングと呼ばれる．この方式ではローディングと外部参照の結合が実行時になってから行なわれる．すなわち，アセンブラはテキスト，結合および再配置情報を原始言語デックから生成し，ローダは主プログラムのみをロードする．主プログラムが外部の番地へ制御を移したり，外部変数（すなわち，この手続きセグメント内で定義されていない変数）を参照しようとするとローダが呼出される．このときに初めて外部参照されたセグメントがロードされる．

この方式の利点は，呼出されたり参照されたりするセグメントが実際に使用されるまでは何らオーバヘッドがないということである．さらに，システムは動的に構成変更が可能である．この方式の主な欠点は，大半の結合処理を実行時に行なうことにしたために生じるかなりの程度のオーバヘッドとシステムの複雑さである．

5.2 絶対型ローダの設計

絶対型ローダの設計を行なって一般的なローダの問題点を示そう．

絶対型ローダ方式ではプログラマおよびアセンブラが割付け，再配置および結合を受持つので，ローダは目的デックのカードを読込み，カードに記されたテキストをアセンブラによって指定された絶対番地に移すことだけを行なえばよい．

目的デックによってアセンブラからローダへ通知しなければならない情報には2種のものがある．第1は，割付けたコアの場所に対してアセンブラが生成した機械命令である．第2は，すべての命令がロードされたときローダが制御を渡すべきプログラムの入口である．この情報がカードで伝達されるとして，

テキスト・カード（命令とデータ）

カードの欄番号	内 容
1	カードの種類＝0（テキス・カードであることを示す）
2	カード上の情報のバイト数（1欄1バイト）
3-5	カード上のデータを置く番地
6-7	空欄（正当性の判定に使用可）
8-72	ロードされる命令やデータ
73-80	カード番号

制御移行カード（プログラムの入口を示す）

カード欄番号	内 容
1	カードの種類＝1（制御移行カードであることを示す）
2	バイト数＝0
3-5	入口の番地
6-72	空 欄
73-80	カード番号

図 5.10 絶対型ローダに対するカード書式

書式の1例を**図 5.10** に示しておく．

図のカード書式で命令はカードに1欄につき1バイトが記入される．8ビット・バイトの可能な 256 とおりの値に各々パンチ・カード符号が対応する．したがって，1枚のカードを読込んだとき，その内容はコアの連続した80バイトに格納される．

絶対型ローダのアルゴリズムは全く簡単である．このローダに対する目的デックは，制御移行カードで終了する一連のテキスト・カードから成っている．したがって，ローダはカードを制御移行カードがくるまで，1枚ずつ読込み，カードで指定された場所にテキストを移す．最後のカードを読込んだとき，アセンブルされた命令はコアの中に入っており，制御移行カードで指定された番地に制御を移すことだけが必要である．この処理の流れ図を**図 5.11** に示す．

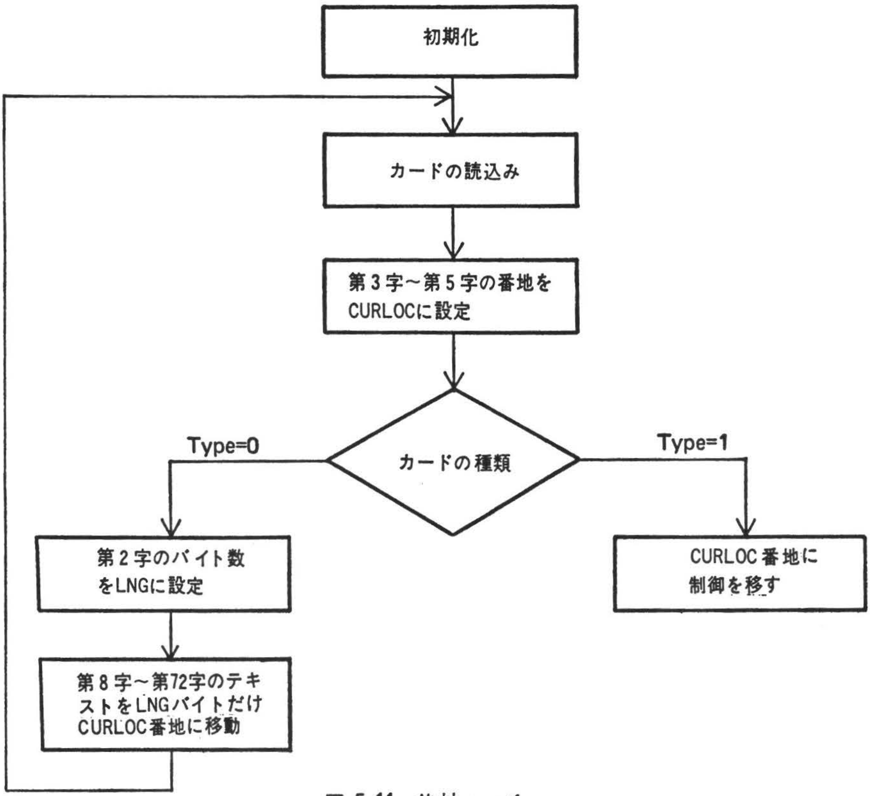


図 5.11 絶対ローダ

5.3 直接結合型ローダの設計

本節では IBM 360 型の直接結合型ローダの設計を述べる．ある種の難解な機能（主として IBM PL/I の組込みとオーバレイ構造に関連したことなど）を省略し，幾つかの形式が考えられるときは最も簡単なもののみ示している．

設計は，アセンブラの設計でとったのと同様の段階を追って行なう（第3章）．なお，直接結合型ローダでは，番地定数の修飾を行なう前にある種の外部記号の（ロード時の）絶対値を知らねばならないことから，2パスの処理が必要である．

5.3.1 問題の設定

IBM 360 の構成は再配置型ローダの行なう仕事を容易にしている．直接番地指定型計算機 IBM 7094 では，ほとんどすべての命令について番地部の再配置が必要である． IBM 360 では命令の再配置はベース・レジスタを用いて行なわれる．ベース・レジスタはアセンブラやローダ以外で設定される．したがって，IBM 360 の再配置型ローダは命令を再配置しないデータ（1語の定数，文字など）と**全く同様に**取り扱うことが可能である．しかし，依然として番地定数は再配置しなければならない．

例えば次の命令は，

```
TEST     START
         USING     *,15
         L         1,DATA
           :
DATA     DC        F'5'
         END
```

次のようにアセンブルされる：

相対番地	**命令／データ**
0	L 1,96(0,15)
⋮	⋮
96	5

プログラムがどこにロードされるかには無関係に，L命令は DATA がプログラムの最初から96バイトの所にある限り不変である．ベース・レジスタ15の内容は，プログラムのロードされる場所によって明らかに変化する．

一方，上の例を次のように変更してみよう：

```
               :
DATA       DC     F'5'
DATALOC    DC     A(DATA)
           END
```

DATALOC は DATA の絶対番地を格納していなければならない．アセンブラには DATA がプログラムの最初から 96 バイトの所にあることだけがわかっている．したがって，ローダはこれにプログラムのロード番地を加えて，

DATALOC に格納されるべき実際の絶対番地を求めなければならない。

ここで**番地定数**の問題の範囲を明らかにしておこう．番地定数には (1)絶対値型，(2)単純再配置型，および (3)複雑配置型がある．

例えば，番地定数 A(LOC1—LOC2) は，

1. **絶対値型**　LOC1 および LOC2 がプログラムの中で定義されている 2 つの再配置型記号のとき．アセンブラはその値を計算することができる．
2. **単純再配置型**　LOC1 がこの手続きの中の再配置を要する記号で，LOC2 が絶対数（例えば LOC2 EQU 5）のとき．アセンブラは相対番地 LOC1 と LOC2 の値の差を計算できるが，ローダはプログラムのロード番地を加えて再配置を行なわなければならない．
3. **複雑再配置型**　LOC1 と LOC2 が他のプログラムのものであるとき．アセンブラは何もすることができず，ローダが値を計算しなければならない．

IBM 360 の直接結合型ローダはアセンブラ，FORTRAN コンパイラ，または PL/I コンパイラによって生成されたプログラムを処理する．元の原始プログラムもアセンブラの記号表もローダには与えられていないことを思い出してほしい．したがって，目的デックは再配置と結合に必要なすべての情報を持っていなければならない．

目的デックには 4 つの部分がある（そして 4 つの対応したカードの書式がある）．

1. 外部記号辞書カード（ESD）
2. テキスト（プログラムの命令とデータ）・カード（TXT）
3. 再配置・結合ディレクトリー・カード（RLD）
4. 終了カード（END）

ESDカードは外部記号辞書，すなわち，記号表を作成するのに必要な情報を持っている．外部記号とはそのサブルーチンの範囲を越えて参照することを認める記号である．原始プログラムの中の普通の記号はアセンブラだけで使用され，それらに関する情報は目的デックには含まれない．

例　プログラムBは NAMES という表を持っているとする．この表はプログラムAから次のようにして参照することができる．

```
A           START
            EXTRN     NAMES
            :
            L         1,ADDRNAME     NAME表の番地を求める
            :
ADDRNAME    DC        A(NAMES)
            END
─────────────────
B           START
            ENTRY     NAMES
            :
NAMES       DC        ───
            END
```

上に示されたように外部記号の型には3種ある：

1. **セグメント定義**（SD）――START または CSECT カードにある名前．
2. **局所定義**（LD）――　ENTRY カードで定義される．この名前は同じプログラムの中になければならない．
3. **外部参照**（ER）――　EXTRN カードで定義される．他のプログラムに同じ名前の ENTRY，START または CSECT カードがなければならない．

各 SD および ER 記号はアセンブラにより番号（1，2，3，……）が打たれている．この番号は記号の**識別番号 ID** と呼ばれ，RLD カードと共に使用される．

TXT カードは，データとそのデータが置かれるべき相対番地のブロックを持っている．ローダがプログラムをロードする場所を決定すると，ローダは**プグラム・ロード番地**（PLA）を相対番地に加えデータをその場所に移す．TXT カードのデータは命令，再配置されないデータ，または番地定数の初期値である．

例

```
相対番地                 命令
  A                      START
                         EXTRN     NAMES
                         USING     *,15
                           :
40                       L         1,ALPHA
44                       BCR       15,14
46                       ----------------   (アセンブラにより
48    ALPHA              DC        F'5'      飛ばされる)
52    ALLOC              DC        A(ALPHA)
56    ADDRNAME           DC        A(NAMES)
                           :
```

生成された TXT カードはつぎのようになる：

相対番地＝40

データ部分＝58 10 F0 48 07 FE *XX* *XX* 00 00 00 05 00 00 00 48
00 00 00 00

データ部分の長さ ＝20バイト

RLD カードは次の情報を持っている：

1. 再配置や結合によって変更されるべき各番地定数の位置と長さ.
2. 番地定数が修飾される（加算または減算）ための外部記号.
3. 行なうべき処理（加算または減算).

RLD カードでは，第 5.1.6 節および**図 5.8** で述べたような実際の外部記号名を使用する代りに，外部記号の識別番号 ID を用いる．この理由は幾つかあるが，最も大きな理由は ID は1バイト長であり，長さ8バイトの記号名を使用するのと比べると RLD カードの容量を大幅に節約することができるからである．しかしこのようにして容量の節約を計ると，後に示すようにローダが複雑になる.

TXT カードの例として上に示したプログラムからは次の RLD カードが作られる：

ID	**フラグ**	**長さ**	**相対番地**
01	＋	4	52
02	＋	4	56

Aの識別番号が1，NAME の識別番号が2とすると，この RLD カードの情報によってローダはAの絶対ロード番地を相対番地 52 の内容に加え，次に NAME の絶対ロード番地を相対番地 56 の内容に加える．

END カードは目的デックの終了を示す．アセンブラの END カードの演算数部に記号が書込まれていれば，これはプログラム全体の実行開始点である．

目的デックの集合には終了を指定するカードが必要である．IBM 360 ローダは普通ローダ終了（LDT）カードまたはファイル終了（EOF）カードを使用する．

サブルーチンA	{	ESD TXT RLD END
サブルーチンB	{	ESD TXT RLD END
サブルーチンC	{	ESD TXT RLD END
		EOF または LDT

図 5.12 は，簡単なプログラムPG1 と PG2 により各種の場合の再配置と結合を説明している．**図 5.13** および**図 5.14** は，それぞれ PG1 と PG2 に対してアセンブラが生成した ESD, TXT および RLD カードを示している．**図 5.15**はプログラムに領域が割付けられ，再配置され，結合され，そしてロードされた後の状態を示している．読者は注意深くこの図を見て，各値が作り出された過程を検証してほしい．

この例につき注目すべき2，3の点は，PG1 と PG2 はA（PG1ENT2−PG1ENT1−3）なる番地定数を持っていることである．**図 5.15** を見ると両方共（番地152，200）この番地定数の値は同じ値（7）である．PG1ENT2 および PG1ENT1 は両方共 PG1 の内部の記号であるので，アセンブラは PG1 を処理しているとき上式の値を計算し値7を決定することができる．**図 5.13** で番地 48〜51 に対する TXT カードは値7を持っているが，この番地定数に対

する RLD カードのないことがわかる．一方，これらの記号は PG2 に対しては外部記号である．したがって，アセンブラは PG2 を処理しているときに番地定数の値を決定することができない．これは**図 5.14** に示されている．相対番地 32〜35 に対する TXT カードは，アセンブラで計算できた番地定数の定数部分の値 −3 を持っている．RLD カードの最後の 2 枚によってローダは，ID 03（PG1ENT2 に対応する）の値を番地 32〜35 の内容に加え，次に ID 02（PG1ENT1 に対応する）の値を差引く．ローダでこの処理が行なわれるとき，PG2 のこの番地定数は PG1 の中のものと同じ値を持つことになる．

直接結合型ローダではある目的デック中の外部参照が，後に出てくる目的デックの処理が終了するまでは計算できないことがあるので，この種のローダでは 2 パスの処理が必要である．これらの機能はアセンブラの 2 つのパスときわめて類似している．直接結合型ローダのパス 1 の主機能は各プログラムにコアの場所を割付け，記号表を作って外部記号の値を記入することである．パス 2 の主機能は実際のプログラム・テキストをロードし，変更の必要がある番地定数の再配置を行なうことである．

原始カード参照番号	相対番地	例題プログラム（原始デック）			
1	0	PG1	START		(a) 手続き PG1
2			ENTRY	PG1ENT1,PG1ENT2	
3			EXTRN	PG2,PG2ENT1	
4	20	PG1ENT1	⋮		
5	30	PG1ENT2	⋮		
6	40		DC	A(PG1ENT1)	
7	44		DC	A(PG1ENT2+15)	
8	48		DC	A(PG1ENT2-PG1ENT1-3)	
9	52		DC	A(PG2)	
10	56		DC	A(PG2ENT1+PG2-PG1ENT1+4)	
11			END		
12	0	PG2	START		(b) 手続き PG2
13			ENTRY	PG2ENT1	
14			EXTRN	PG1ENT1,PG1ENT2	
15	16	PG2ENT1	⋮		
16	24		DC	A(PG1ENT1)	
17	28		DC	A(PG1ENT2+15)	
18	32		DC	A(PG1ENT2-PG1ENT1-3)	
19			END		

図 5.12 例題の手続き PG1 および PG2

パス1はすべてのセグメントに記憶場所を割付け，記号表の中にすべての外部記号の値を格納する．これらの外部記号は，他のプログラムの局部定義 ESD カードによって与えられる．すなわち外部参照の行なわれたすべての記号に対しては，他のプログラムの中でこれが定義されていなければならない．ローダは記号表の中にすべての外部記号の絶対番地を記入する．パス2でローダは，テキストを割付けられた場所に置き，再配置定数を変更して再配置を行なう．図5.16は直接結合型ローディング方式のパスの関係を示している．

5.3.2 データ構造の規定

ESDカード

原始カード参照番号	名 前	型	*ID*	相対番地	長 さ
1	PG1	SD	01	0	60
2	PG1ENT1	LD	——	20	——
2	PG1ENT2	LD	——	30	——
3	PG2	ER	02	——	——
3	PG2ENT1	ER	03	——	——

TXT カード
(関係あるもののみ，すなわち，番地定数を含むもの)

原始カード参照番号	相対番地	内 容	註
6	40-43	20	
7	44-47	45	= 30 + 15
8	48-51	7	= 30-20-3
9	52-55	0	PG I には未知
10	56-59	-16	= -20 + 4

RLD カード

原始カード参照番号	*ESD ID*	長 さ *(バイト)*	フラグ +または−	相対番地
6	01	4	+	40
7	01	4	+	44
9	02	4	+	52
10	03	4	+	56
10	02	4	+	56
10	01	4	−	56

図 5.13 プログラム PG1 の目的デック

設計手続きの第2段階はローダの各パスで必要とされるデータ・ベースを示すことである.

パス1のデータ・ベース:

1. 目的デックの入力.
2. プログラム・ロード開始番地(IPLA)——プログラマやオペレーティング・システムから指定される. 第1セグメントをロードすべき番地を示す.
3. プログラム・ロード番地 (PLA) カウンタ——各セグメントに割付けられた場所を記憶しておくために使用される.
4. 汎用外部記号表 (GEST)——外部記号とそれに対応したコアの番地を格納する.
5. パス2で使用するための入力の写し——磁気テープ, ディスク, ドラムなどの補助記憶装置. パス2で元のカード・デックを再び読込んでもよい.

***ESD* カード**

原始カード参照番号	名 前	型	*ID*	*ADDR*	長 さ
12	PG2	SD	01	0	36
13	PG2ENT1	LD	—	16	—
14	PG1ENT1	ER	02	—	—
14	PG1ENT2	ER	03	—	—

***TXT* カード**
(関係あるもののみ)

原始カード参照番号	相対番地	内 容
16	24-27	0
17	28-31	15
18	32-35	-3

***RLD* カード**

原始カード参照番号	*ESD ID*	長 さフラグ (バイト)	フラグ +または−	相対番地
16	02	4	+	24
17	03	4	+	28
18	03	4	+	32
18	02	4	−	32

図 5.14 プログラム PG2 の目的デック

6. **ロード・マップ**の印刷出力――各外部記号とそれに割付けられた値を示す.

パス2のデータ・ベース：

1. パス1に入力された目的プログラムの写し.
2. プログラム・ロード開始番地 (IPLA).
3. プログラム・ロード番地カウンタ (PLA).
4. 汎用外部記号 (GEST)――パス1で用意され，各外部記号とそれに対応

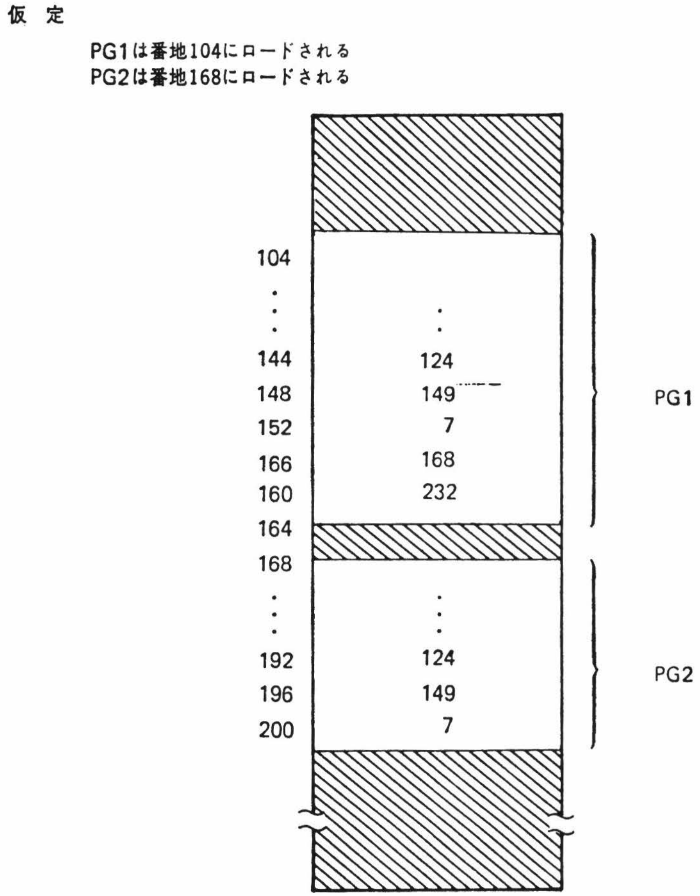


図 5.15 プログラム PG1 および PG2 をロードした後の主記憶

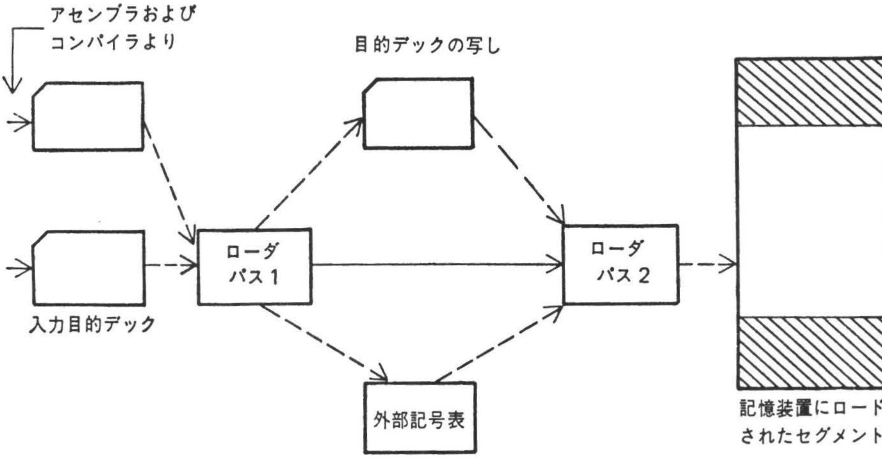


図 5.16　2パス直接結合型ローダ方式

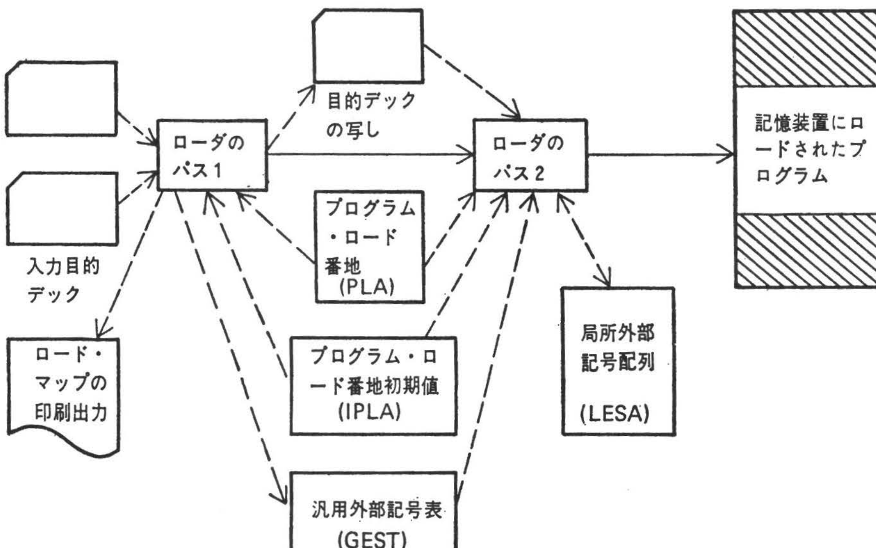


図 5.17　ローダのパスとデータ・ベースの関係

した絶対番地の値を格納している．

5. 局部外部記号列 (LESA)——ESD および RLD カードで用いられている

欄	内 容
1	16進バイトX'02'(カード穿孔位置12-2-9)
2-4	ESDという文字
5-14	空 白
15-16	プログラム名(SD)または外部記号(ER)に対するESDの識別番号(ID) (ID はプログラム中で一意的な値をとる), または, 入口(LD)に対しては空白. 次の箱を参照
17-24	名前, 余白は空白
25	ESDの型

型	16進コード
SD	01
LD	02
ER	03

欄	内 容
26-28	相対番地または空白 (ADDR), 下の箱を参照
29	空 白
30-32	プログラムの長さまたは空白
33-72	空 白
73-80	カード番号

ESD形式と規約

	型	ID番号	*ADDR*	長 さ
プログラム名(セグメント定義)	SD	01	0	プログラムの長さ
入口(局所定義)	LD	—	相対番地	—
外部参照	ER	一意的な番号	—	—

(ID番号は通常順番に割当てられる)

図 5.18 ESD カードの書式

ESD の ID 番号と対応した外部記号の絶対番地の値.

5.3.3 データ・ベースの書式

設計手続きの第3段階はデータ・ベースの書式と内容を規定することである. 主なデータ・ベースは**図 5.17** に示してある.

目的デック： 目的デックについてはこれまで何回も説明した. **図 5.18, 図 5.19, 図 5.20** および **図 5.21** は各種の IBM 370 や 360 の直接結合型ローダに使用されている実際のカード・デックの詳細を示している. 特定の機種ということは問題でなく, 情報を符号化する方法の良い例として示している.

欄	内容
1	16進バイトX'02'
2-4	TXTという文字
5	空 白
6-8	第1データ・バイトの相対番地
9-10	空 白
11-12	バイト・カウント(BC)＝17－72欄の中の情報バイト数
13-16	空 白
17-72	1から56字のデータ・バイト(命令と"データ"は同じように見える)
73-80	カード番号

図 5.19 TXT カードの書式

欄	内容
1	16進バイトX'02'
2-4	RLDという文字
5-18	空 白
19-20	ESDカードのSDまたはERに割当てられたID番号
21	フラグ・バイト(下の箱を参照)
22-24	番地定数の第1バイトの相対番地(ADDR)
25-72	空 白
73-80	カード番号

フラグ・バイトの規約

ビット	
0-3	未使用
4-5	番地定数のバイト長 00＝1バイト 01＝2バイト 10＝3バイト 11＝4バイト
6	0は番地定数にESD番地を加算することを示す 1は番地定数にESD番地を減算することを示す

図 5.20 RLD カードの書式

汎用外部記号表： 汎用記号表（GEST）は，外部辞書（ESD）カードで定義されているセグメント定義（SD）や局所定義（LD）の外部記号を格納するために使用されている．外部記号がパス1において見付かると，これらに絶対コア番地が割付けられる． この番地は**図 5.22** に示したようにして記号と共に GEST に格納される．

読者はアセンブラの設計に関して第3章で述べた記号表と探索／分類法を復習してほしい． GEST はアセンブラの 記号表と同様に汎用の性質を 持ってい

欄	内 容
1	16進バイトX'02'
2-4	ENDという文字
5	空 白
6-8	プログラムの最初からでないとき、アセンブラのENDカードで指定された実行開始点(ADDR)
9-72	空 白
73-80	カード番号

図 5.21 END カードの書式

← 各欄 12 バイト →

外部記号 (8バイト) (文 字)	割当てられたコアの番地 (4バイト) (10進)
"PG1bbbbb"	104
"PG1ENT1b"	124
"PG1ENT2b"	134
"PG2bbbbb"	168
"PG2ENT1b"	184

註：このGESTの例は図5. 12および図5. 15の例に基づいている.

図 5.22 汎用外部記号表 (GEST) の書式

る.

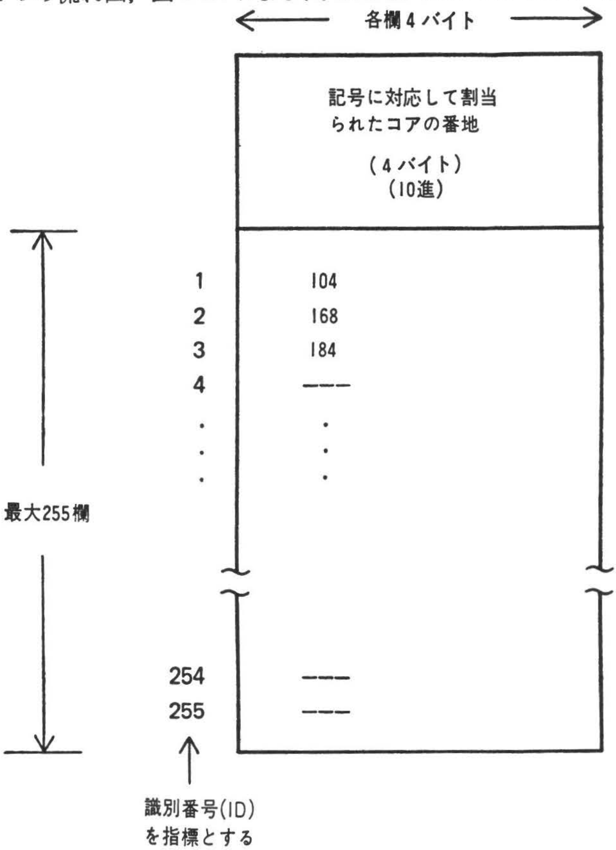
局所外部記号配列： 先に述べたように，再配置や結合のために使用される外部記号は，RLD カードでは，記号名を用いずに，ID 番号によって識別されている. ID 番号は SD または ER 指定のある ESD カードの1つと一致しなければならない. この技法は RLD カードの容量を節約し，汎用外部記号表を何回も探索せずにすむようにして処理速度を向上させる.

このためには RLD カードの ID 番号と絶対コア番地の対応を付けることが必要である. ESD カードは ID 番号とそれに対する記号名を納めている. 一方この記号と絶対コア番地との関係の情報は GEST から求められる. ローダのパス2において個々の目的デックの GEST と ESD の情報が合併されて，直接 ID 番号と番地の値の関係を示す局所外部記号配列 (LESA) が作り出される. 原理的には各セグメント毎に別の LESA を作る必要があるが，LESA

は一時に１個しか作られないので，同じ配列の領域を各セグメントで再使用できる．**図 5.23** に局所外部記号配列(LESA)の書式を示している．なお，GEST とは異なって LESA を探索する必要はない．ID 番号が与えられると対応した値は，LESA(ID) と書けるので直ちに求めることができる．

5.3.4　アルゴリズム

次の２つの流れ図，**図 5.24** および**図 5.25** は IBM 360 型計算機の直接結合



註：このLESAの例は図5.12，図5.13および図5.15に
示されたPG１セグメントに基づいている。

図 5.23　局所外部記号配列 (LESA) の書式

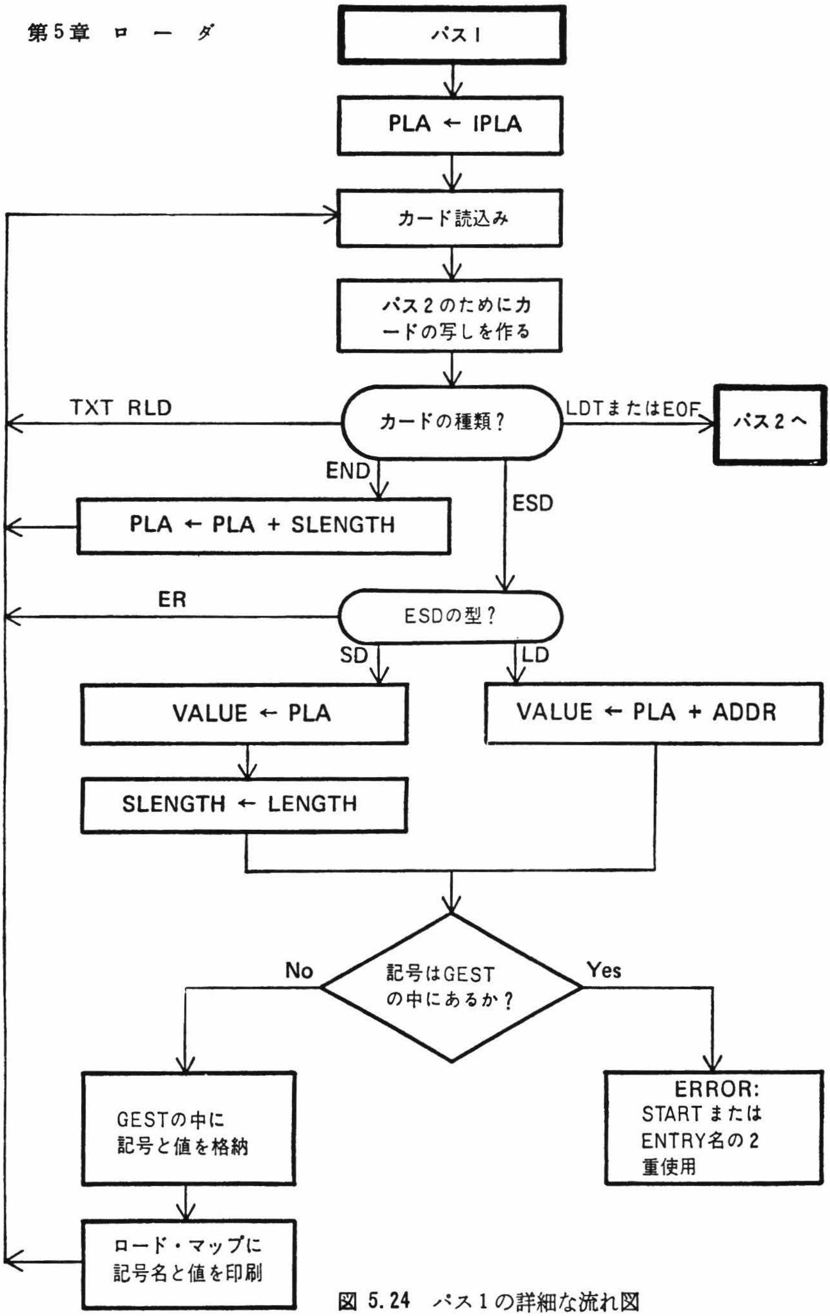


図 5.24 パス1の詳細な流れ図

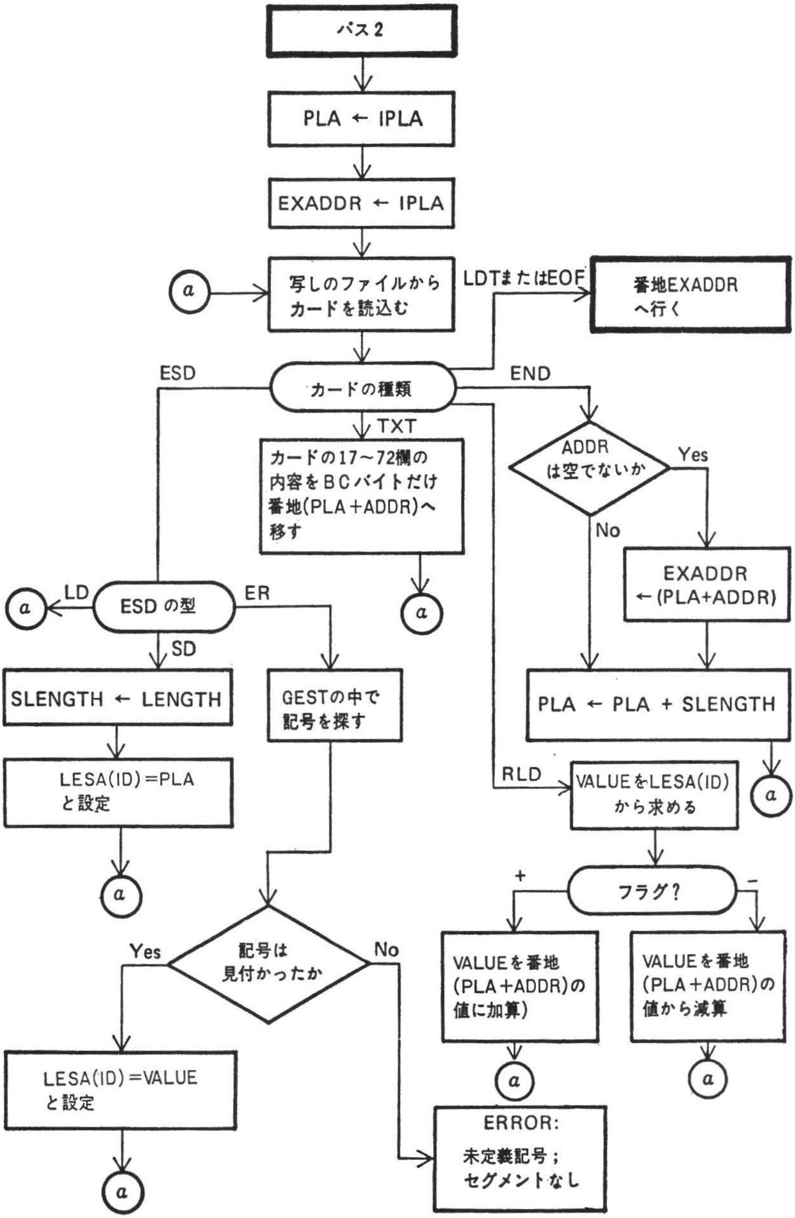


図 5.25 パス2の詳細な流れ図

型ローダのアルゴリズムを示している．これらの図はほとんどすべての論理操作を含んでいるが，複雑なローダによって実行される処理を簡略化して示している．特に，COMMON セグメント，ライブラリ処理，ダイナミック・ローディング，およびダイナミック・リンキングなどの“特殊”機能は含んでいない（これらの多くは演習問題と後の章とで論じられている）．

パス1――セグメントの割付けと記号の定義： パス1の目的は各セグメントに格納場所を割付け，それによってすべての外部記号の値を定義することである．プログラム全体で必要となる記憶容量を最小にしたいので，各セグメントには前のセグメントの直後に空いている場所を割付けることにする．ローダにとっては，最初のセグメントをどこにロードできるかを知ることが必要である．この番地はプログラム・ロード開始番地（IPLA）として，通常，オペレーティング・システムによって決定される．ある種のシステムではプログラマが IPLA を指定することができる．いずれにせよ，IPLA はローダに供給されるパラメータであると考えておく．

最初プログラム・ロード番地（PLA）はプログラム・ロード開始番地（IPLA）に設定される．目的カードが1枚読込まれ，パス2で使用するために写しが作られる．読込まれたカードは ESD，TXT，RLD，END または LDT/EOF のいずれかである．それが TXT または RLD カードであればパス1では何も処理することがないので次のカードが読込まれる．ESD カードは外部記号の型 SD，LD または ER によってそれぞれ異なった処理を受ける．セグメント名を定義する SD 記号の ESD カードが読込まれたとき，長さの部分 LENGTH はいったん変数 SLENGTH に保存される．この記号（セグメント名）の値 VALUE としては PLA の値が割付けられる．次に記号とその記号に割付けられた値は GEST に格納される．既に GEST の中に記号が存在すれば同じ名前の SD または LD 型の ESD カードが前に存在したことになり，これは誤りである．記号とその値はロード・マップの一部として印刷される．同様の処理が LD 記号についても行なわれる．この記号に割当てられる値は PLA の値と ESD カードで示された相対番地 ADDR とを加えた値である．ER 記号はパス1では何も処理することがない．END カードがくると，プログラム・ロー

ド番地（PLA）は，SLENGTH に保存されていたセグメントの長さだけ増加され，次のセグメントに対するPLA の値となる．LDT または EOF カードが最後に読込まれるとパス1は完了し制御はパス2に移される．

パス2——テキストのロードと番地定数の再配置／結合： パス1ですべてのセグメントに記憶場所が割付けられ，外部記号が定義された後に，テキストをロードし，番地定数を調整（再配置または結合）してローディングを完了することができる．パス2の終りでローダはロードされたプログラムに制御を移す．次の簡単な規則が実行開始点を決定するために使用される：

1. END カードで番地が指定されていれば，この番地が実行開始番地として使用される．
2. その他の場合には第1セグメントの始めより実行を開始する．

プログラム・ロード番地（PLA）はパス2の最初でパス1と同様に IPLA の値で初期化され，実行開始番地（EXADDR）も IPLA の値で初期化される．パス1で作られた目的デック・ファイルからカードが1枚ずつ読込まれ，カードの5つの型にしたがって次のような処理が行なわれる：

ESD カード

ESD カードのそれぞれの型は次のように処理される．：

SD 型 ESD： セグメントの長さ LENGTH は，いったん変数 SLENGTH に保存される．局所外部記号配列の適当な欄 LESA(ID) がプログラム・ロード番地の値に等しく設定される．

LD 型 ESD： LD 型 ESD はパス2では何も処理することがない．

ER 型 ESD： 汎用外部記号表（GEST）を探索して ER 記号と一致するものを求める．もし見付からなければ対応したセグメントや入口は存在せず，誤りである．GEST に記号が見付かれば，その値が取り出されて局所外部記号配列の欄 LESA (ID) に記入される．

TXT カード

TXT カードが読込まれたとき，テキストはカードから再配置すべきコアの

場所（PLA＋ADDR）に複写される．

RLD カード

再配置および結合のために使用される値が局所外部記号配列の ID 番目の欄 LESA(ID) から取り出される． フラグの指定（＋または－）にしたがってこの値は処理すべき番地定数に加えたり，差引いたりされる．処理すべき番地定数の値は実際に再配置されている番地 PLA と RLD カードの ADDR 部の和として求められる．

END カード

END カードに実行開始番地が指定されていれば， それを PLA で再配置した後に変数 EXADDR に保存する．プログラム・ロード番地は SLENGTH に保存されたセグメントの長さだけ増加されて， 次のセグメントに対する PLA とされる．

LDT/EOF カード

ローダはロードされたプログラムの実行開始番地として EXADDR に指定された番地に制御を渡す．

5.4 まとめ

ローダの4つの基本的な機能は割付け，結合，再配置，およびローディングである．各種のローダ（例えば“コンパイル・実行”型，絶対型，再配置型，直接結合型，ダイナミック・ローディング型，およびダイナミック・リンキング型）は主としてこれらの4つの基本機能が実行される方法に相違がある．

典型的な直接結合型ローダは2つのパスが必要である．パス1はセグメントに領域を割付け，外部記号の値を定義する．パス2はテキストをロードし，パス1で作られた汎用外部記号表を使用して番地定数を再配置したり結合したりする．

目的は全く異なっているが，直接結合型ローダの設計はアセンブラの設計と多くの類似点を持っている．特に，記号表の使用は両方の場合共に重要である．

演 習 問 題

1. 次に示す型の番地定数の例を示せ.
   a. 単純再配置型
   b. 絶対値型
   c. 複雑再配置型

2.

```
PROG    START   0
        ENTRY   PLACE
        EXTRN   SIN
ONE     EQU     1
          :
LOOP    ---
PLACE   ---
          .
KOHN    ---
          :
        END
```

上のプログラムを参照して下の番地定数の型, 絶対値型, 単純再配置型, または複雑再配置型を区別せよ. また各々の番地定数がどのようにして計算されるかを説明せよ. 目的デックを示す必要はない.

   a. A(SIN)
   b. A(KOHN−LOOP+ONE)
   c. A(PLACE)
   d. A(SIN−LOOP)

3. a. 本章で論じられた幾つかのローディングの方式では, 結合はプロセスの生涯の様々な時点で行なわれる. “結合”という言葉を定義し, その例を示せ.
   b. 次のローディング方式においてはいかなる時点に結合が行なわれる

か.

1) BSS ローダ

2) 直接結合型ローダ

3) 絶対型ローダ

4) ダイナミック・バインダ

5) ダイナミック・リンキング・ローダ

6) オーバレイ

7) リンケージ・エディタ

c. 1) アセンブル時に結合するよりロード時に結合することの利点を2つあげよ.

2) また欠点を2つあげよ.

d. 1) ロード時に結合するより実行時に結合することの利点を2つあげよ.

2) また欠点を2つあげよ.

4. RLD, ESD, TXT および END カードの各々の機能を述べよ.

5. 目的プログラム・デックの中のRLD, ESD, TXT および END カードの順番はどうなっているか. なぜそうなっているのか.

6. ESD カードに示してある ID 番号の目的は何か. 局所的に定義されている記号にはなぜ不要なのか.

7. IBM 360 アセンブラ言語から番地定数を取り除いたとする.

a. 目的デックのどの部分を簡単にしたり省略したりできるか.

b. ローダはどのように簡単になるか.

8. 次の3つの簡単なプログラムを考えよ:

```
PGA      START
         ENTRY    PGA1
         EXTRN    PGB, PGC, PGC2
         DC       A(PGA), A(PGB + 4)
PGA1     DC       A(PGA1 - PGA), A(PGC2 - PGC)
         END

PGB      START
         ENTRY    PGB1
         EXTRN    PGA, PGC1
PGB1     DC       A(PGC1 - 4), A(PGB1)
PGB2     DC       A(PGB + 4), A(PGB1 - PGB)
```

```
PGB3      DC        A(PGC1 + PGB - PGA - 16)
          END

PGC       START
          ENTRY     PGC1, PGC2
          DC        A(PGC2 - PGC)
PGC1      DC        A(PGC1 - 4), A(* + 4)
PGC2      DC        A(PGC + PGC2 - PGC1)
          END
```

a. これらの3つのプログラムは10進の400番地からPGA, PGBおよびPGCの順でロードされるものとする．汎用外部記号表（GEST）に各記号を記入せよ．（ヒント：各プログラムは2語境界から開始しなければならないことを思い出せ.）

記　号	絶対番地
PGA	400
PGA1	
PGB	
PGB1	
PGC	
PGC1	
PGC2	

b. なぜ記号PGB2およびPGB3はGESTにないのか.（1～2文で答えよ.）

c. 3つのプログラムがロードされ結合された後の記憶装置の内容を示せ.

9. 本節では2パス直接結合型ローダを論じた.

a. パス1の機能は何か.

b. パス2の機能は何か.

c. 1パス・ローダに限定されるとすれば，利用者にはどのような機能が使用できるか.

例えば，

番　地	内　容
400	
404	
408	
412	
416	
420	
424	
428	
432	
436	
440	
444	
448	
452	
456	
460	

1)　単純な番地再配置はどうか.

2)　外部記号はどうか.

3)　その他.

その理由を示し，課せられる制限事項を述べよ.

d.　1パス・ローダのためのアセンブラの出力カードを定めよ．これらのカードにはどのような情報が記されるのか（例えば，ESD，RLD，TXT，END に対応するカード).

e.　そのローダの流れ図を描け.

10. a.　次頁に示す目的デックが与えられたとする．できるだけ記号を用いて元の原始デックを作れ.

b.　解は1つに限るか．その理由は何か.

11. 絶対型ローディングを行なうコードを作るために，新しい擬似命令がアセンブラに追加された．この擬似命令は命令コードとして ABS を持ち，また

ESD

記号	型	ID番号	相対番地	長さ
PGM	SD	01	0	92
ENT1	LD	–	20	--
ENT2	LD	–	40	--
SQRT	ER	02	--	--
TAN	ER	03	--	--
COS	ER	04	--	--

TXT

相対番地	値
80	-40
84	0
88	20

RLD

相対番地	+/-	ID番号
80	+	02
80	+	03
80	-	01
84	+	04
88	+	01
88	-	02

プログラムのロードを開始する番地を指定する1つの演算数を持っている．アセンブラはこの擬似命令によって絶対型ローダの入力コードを生成する．

a. ABS カードは入力デックのどこに置かれなければならないか． すなわち， ABS カードの前に置けるものは何で， 後に置かねばならないものは何か．

b. ABS 擬似命令はアセンブラのどの パスで処理されるか． その理由は何か．

c. ABS モードのときアセンブラのどのデータ・ベースが影響を受けるか． 変更点を具体的に 理由と共に示せ（アセンブラは DLL（直接結合型ローダ）のためにコードを正しく生成するものと考える.）

d. 記号表の中の再配置／絶対値フラグを使用する必要があるか．その理由は何か．

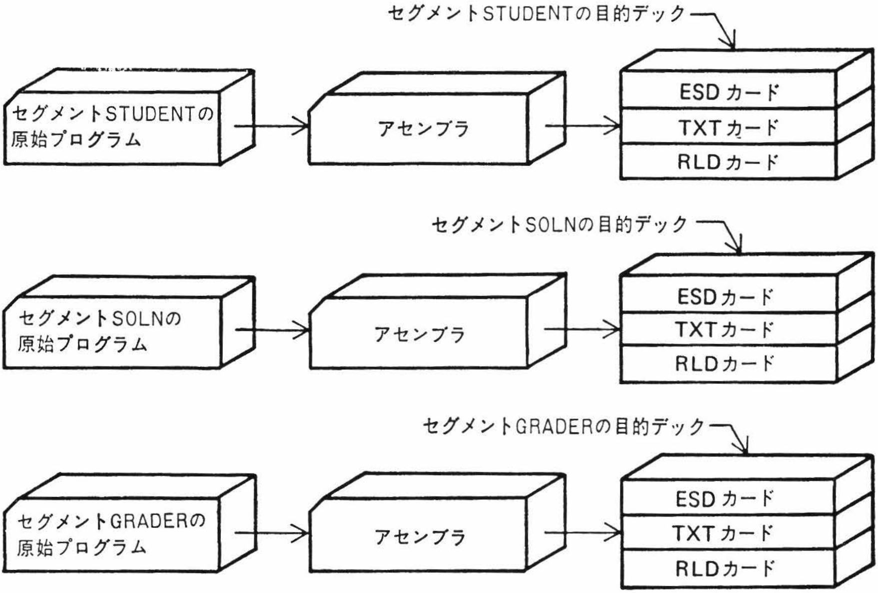
12. 外部参照を行なわない1つのセグメントをロードする簡単な再配置型ローダの設計を行なえ．計算機は非常に小規模であり余分の情報のための余裕はないので真に必要な情報のみを含むように設計しなければならない．互換性を保つために**図 5.10** に示された絶対型ローダ・カードを用いて設計を始めよ．

a. 絶対型ローダ・カードを変更してもよいし，必要ならば付加的な情報を加えて新しいカードを設計してもよい．しかし不要な情報を含めてはなならない．異なった書式のカードを設計するのをためらう必要はない．アセンブラは新しい設計によるカードを生成するように設計変更することができる．

b. 目的デックがどのように構成されるべきかを示せ．例えば，どのカードが最初に作られ次に何がくるかなど．

13. 後で使用したりローディング時間を節約して効率を向上させるために，幾つかのセグメントを1つのセグメントに結合することがしばしば有効である．

a. セグメント SOLN および GRADER の目的プログラムを原始プログ



ラムから作り上げよ．まず，セグメント STUDENT の例を示す．

例

1）セグメント STUDENT の原始プログラム

```
STUDENT   START   0
          ENTRY   A
          EXTRN   SOLN,DELTA
          BALR    15,0
          USING   STUDENT +2,15
          SR      4,4
          L       4,TWO
          L       5,TWO
          ST      5,SAVE
          BR      14
          DC      5F'01'
          DC      A(A+10)
SAVE      DC      12C'0'
A         DC      A(DELTA)
TWO       DC      F'2'
          DC      A(STUDENT-SOLN)
          END
```

2）セグメント STUDENT の目的プログラム

	記号	型	ID番号	相対番地	長さ
ESD	STUDENT	SD	01	0	68
	A	LD	--	56	--
	SOLN	ER	02	--	--
	DELTA	ER	03	--	--

	命令コード	相対番地	番地定数	番地定数の　値
TXT	BALR	0	--	--
	SR	2	--	--
	L	4	--	--
	L	8	--	--
	ST	12	--	--
	BR	16	--	--
	DC	20	--	--
	DC	40	A(A+10)	66
	DC	44	--	--
	DC	56	A(DELTA)	0
	DC	60	--	--
	DC	64	A(-SOLN)	0

	ID番号	フラグ (長さ)	フラグ (+−)	相対番地
RLD	01	4	+	40
	03	4	+	56
	02	4	-	64
	01	4	+	64

例で示されたようにしてセグメント SOLN および GRADER の目的デックを次の原始プログラムから作れ.

セグメントSOLN の原始プログラム

```
SOLN     START   0
         EXTRN   A,C2
         BALR    15,0
         USING   *,15
         SR      4,4
         SLL     1,2
         L       0,ANS(1)
         BR      14
         DC      A(A+10)
CODE     DC      2C'JJ'
         DC      A(C2)
         DC      12C'2'
ANS      DC      5H'0'
         END
```

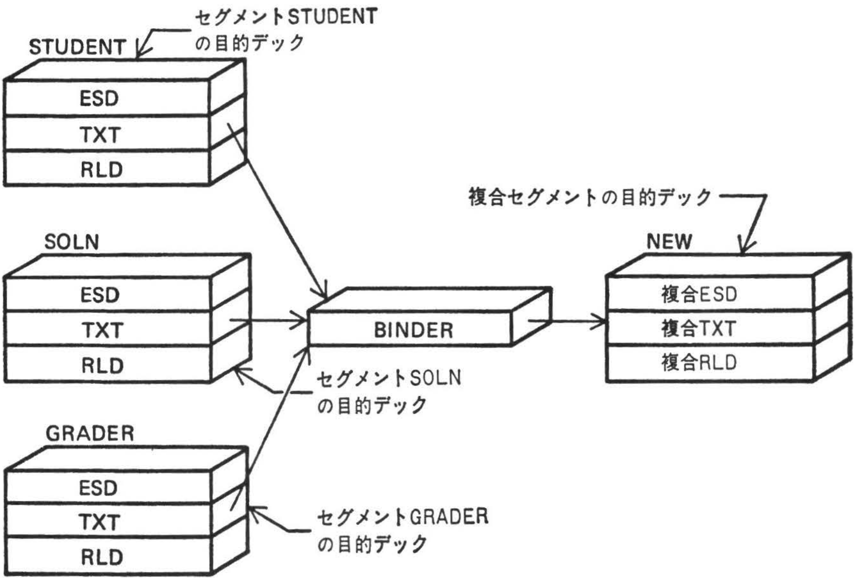
セグメントGRADER の原始プログラム

```
GRADER   START   0
         ENTRY   GO,C2
         BALR    15,0
         USING   *,15
         SR      4,4
         L       6,C3
         ST      1,C3+4
         LA      1,C3
         L       2,JOHNSN
         BR      14
C2       DC      A(GRADER)
C3       DC      8F'3'
         END
```

b. セグメント STUDENT, SOLN および GRADER の目的デックを1つの複合セグメントに結合し，そのセグメントについて幾つかの問に答えよ．本書に述べられている直接結合型ローダ (DLL) を使用することができるように，複合セグメントは元のセグメントと同じ書式で作られていなければならない．したがって複合セグメントは次に示されたような形となる.

複合 ESD は複合されたセグメントの中の各名前に対して1つの欄を持ち，さらに，新しいセグメント名，この場合 “NEW”，のための欄が追加されている.

複合テキストにはバインダ BINDER に与えられた順(すなわち，STU-



DENT，SOLN，GRADERの順）で複合されたセグメントのテキストが置かれている．セグメントは2語境界から始まらなければならない．複合 RLD はすべての RLD カードを合併したものであり，複合 ESD および複合テキストを参照している．

問

1) セグメントを複合するには各セグメントの3つの部分を別々に合併しなければならない．複合テキストはいつ作られるか．テキスト・カードを修正する必要はあるか．その理由は何か．
2) ESD 部はただ1つの SD カード（“NEW”という名前）だけを持つようにすることができる．元のセグメントの SD はどのようにすればよいか．セグメント STUDENT は記号 SOLN を参照しているからこれを無視できないことに注意せよ．
3) 複合 RLD が作られるとき，RLD カードのどの部分を修正せねばならないか．また，どのように修正されるのか．
4) 例として示された複合セグメントのどの RLD カードが依然として外部記号を参照しているか．その記号は何か．

5) 例に示されたセグメント STUDENT の目的デックと同じ形で複合セグメントの内容を記せ．**註**：番地定数を参照しない複合テキストの行をすべて省略してもよい．

6) 複合セグメントの中で TXT カードが持つ番地定数の実際の値を示せ．例えば，セグメントSTUDENTの中の定数DC A(DELTA)の値は，相対的なオフセットの指定はなく，DELTA は STUDENT に対しては外部記号であるので，0 である．

7) 複合セグメントを作るプログラムには幾つのパスが必要か．理由を示して答えよ．

8) BINDER に必要となるデータ・ベースは何か．

9) BINDER と DLL はどこが違っているか．答えは2，3行でよい．

10) 複合 TXT の中で同じ番地を参照している所，例えば DC A(A＋10)，をいずれか1つを残して他を消去することは可能か．理由を述べよ．また可能なときはその方法を示せ．この方法は名案だと思うか．

14. 互いに参照し合うセグメントを結合しておくことは，ローディング時間を節約するのにしばしば有効である．デバックするときには，これらのセグメントの虫取りを行なうために再び結合を外すことができると (unbind, delink) アセンブル時間を節約することができて好都合である．しかしながら，複合セグメントの結合を外すことは不可能である．その理由は SD カードや ER カードなど元のセグメントに存在していた情報は，直接結合型ローダが複合セグメントをロードするためには不要となり，結合の結果捨てられてしまっているからである（問5.13参照）．

ABC 会社は，セグメントの結合を外すことができるようにするために，宿題程度で最も簡単だと考えられるバインダの書式変更を行なうことにし設計基準を定めた．しかしながら，まだ幾つかの設計上の問題点が残されている．どれだけ余分の情報がどのカードに必要かを決定してこの問題を解決したい．

a. 元の SD を新しく LD にするときセグメント長がなくなることに注

意して次に答えよ．

1) この部分が残されていれば結合解除に有用か．

a) ESD カードはどうするか．

b) TXT カードはどうするか．

c) RLD カードはどうするか．

2) この部分がなくなるとして，各セグメントの元の長さは ESD に残っているすべての情報から計算できるか．その理由は何か．

3) 上の答えから，セグメント長の部分は結合解除に必要か．

b. 元の ER と元の LD を結合するときに，すべての ER は捨てられ，LD は ESD に入れられていることに注意せよ．結合を解除するためには LD は元の文と対応をとり，すべての ER を再び作り出し，元の文と対応を付けなければならない．このことは，もともと LD であった LD に対して新しいデータの欄を作らねばならないことを示している．1つは FROM と呼ばれる固定長の欄であり，この LD が作られたセグメントの ID を格納する．もう1つは IN と呼ばれる可変長の欄で，この欄には ER が存在したセグメントの ID を格納する．

1) 元の LD に対してこれら2つのデータ部分を考えよ．FROM 部は必要か．IN 部は必要か．その理由は何か．

2) ER について考え，問1)に答えよ．

c. 完成した LD カードに新しいデータを追加しないとしたら，バインダはどのような基準によって，元の SD と元の LD とを区別するか．LD カードの各データの内容だけから区別を行なうものとして答えよ．

15. 独立のプログラム・バインダと簡単なローダを用いるシステムを考えてみよう．

a. どの機能を DLL から取り除き，その代りにバインダで行なうことができるか．ローダをできるだけ簡単にするように考えよ．

b. どの機能は簡略化されたローダによっても果さねばならないか．

c. 簡略化されたローダには幾つのパスが必要か．その理由何か．

d. これまで論じたローダのすべての方式を考えてみて，今作ったローダはどれに相当するか示せ．

第 6 章

プログラム言語

第6章，第7章および第8章は高水準のプログラム言語について述べる．第6章の目的は次の3つである：

1. 高水準言語の重要な特徴を示すこと．
2. これらの特徴のシステム・プログラミングへの応用を簡単に議論すること．
3. コンパイラの設計のために，例として，言語 PL/I を紹介すること．

PL/I を原始言語として選んだ理由は，多くの言語は PL/I のサブセットであり，さらに単に教育用の目的で作られたものでなく実際に使用されている言語だからである．PL/I[1] はデータ構造やストリング処理，ポインタ，記憶域のクラス，条件処理といった高級な機能を備え，将来の言語が当然備えるべきものを持っている．これらの機能はシステム・プログラミングに重要であるし，さらに，これらはコンパイラの設計者に難しい問題を与えている．

本章は主として，FORTRAN や ALGOL，COBOL といった初期の高水準言語を既に知っている読者のために書かれている．PL/I や ALGOL 68 などの新しい言語で発展した拡張機能について述べる．読者はこれらの拡張を各種のマニュアルや参考書によって学んでほしい．

第7章においては高水準言語の形式的な表現を与え，この分野における理論的な研究成果を述べる．高水準言語のためコンパイラ設計は第8章に述べる．

1 科学用に設計され，提案されている FORTRAN Ⅷ という名の進歩した FORTRAN Ⅳ および事務用に設計された COBOL の最新版との関連において，この新しい PL/I 言語は Ⅷ·BOL と呼ぼうという話があった．

6.1 高水準言語の重要性

プログラミングにおいて高水準言語はとみに重要になってきている．これらの言語は，これまでアセンブラ言語の本領であると考えられていた仕事までをも効果的に遂行するために使用されている．例えば MIT のプロジェクト MAC における MULTICS タイムシェアリング・システムや Burroughs の新しいシステムは，それぞれ主として高水準言語 PL/I または ALGOL によって書かれている．

このような言語を使用することの利点を論じよう．幾つかの点は自明ではなく，恐らく議論の余地の多いものである．読者はこれらの詳細と例について第10章に掲げた文献を参照するとよい．

高水準言語を使用することによる利点は次のとおりである：

1. より少ない人間と，より少ない生産管理でよい

種々の研究によると，1人のプログラマが作成することのできるプログラムの行数は，デバッグまで含めて比較すると，使用する言語に関係なしにほぼ一定である．高水準言語はしばしばアセンブラ言語より10倍も表現能力が大きい（すなわち，高水準言語の1つのステートメントと同じことを行なうのにアセンブラ言語では10ステートメントが必要である)．したがってより少ない人間とより少ない管理で仕事を行なうことができる．さらに，より小さい作業班で仕事ができることはしばしば能率をきわめて大きく向上させる効果がある．

2. 修得期間が短い

修得期間がきわめて短いので，新しいスタッフが計画に直ちに参加して仕事を行なうことができる．プログラムやモジュールは簡潔でそれを読む人にとってもわかりやすい．

3. デバッグ機能が改善されている

機械語の誤りやトリック（ビット処理などの論理表現の手段）よりも，むしろ，設計における論理の誤りや欠陥といったことに注意を集中できる．

4. 優れた文書化能力

PL/I プログラムはそれ自身かなりの程度文書化されたものである．例えば，1つのモジュールの記述は PL/I のデータ構造とこのデータを処理する短いステートメントで行なうことができる．たいていのオペレーティング・システムのモジュールは複雑なデータ構造を処理する単純なプログラムにすぎないということがいわれている．

5.　ハードウェアから独立している

PL/I のような高水準言語は，機械語に比べてハードウェア依存度がきわめて小さい．このことにより機械が異なっても同じプログラムを実行することができる．最も重要なことは，プログラマは“ビットをいじくる”ことより“設計”に専念できることである．

6.　目的プログラムの効率改善の可能性

高水準言語を使用する上で最も大きな論点となるのは目的コードの効率であろう．微視的に見たとき（ステートメント毎に比較すると）上手な機械語のプログラマはコンパイラよりも効率の良いコードを作ることができる．しかしながら，大量のコードから構成されるシステムにおいて効率の問題は自明な問題ではない．熟練した機械語のプログラマが PL/I プログラムをコーディングし直すと速度や容量で2～3倍も能率が良いものを作れることもある．しかしながら，1バイトとか1語とかいった節約が巨視的な効率を考えたとき真の節約になるかということはいささか疑わしい．重要なことは処理の論理が本当に能率の良いものであるかを調べることである．高水準言語を使用したときには，処理の論理を巨視的に見て，問題点をより容易に調べることができる．これらの点は，機械語のプログラマにはプログラムを徹底的に最適化する時間の余裕がほとんどないという事実に照して考慮されなければならない．

7.　システムをより能率的に働かせること

普通ソフトウェアの効率の低さを予想することはできない．最終的には，速度や効率が重要であることが判明したセグメントをコーディングし直すことがよく行なわれる．大きなシステムにおいて，実行時間の 80% が容量において 20% 以下のコードによって費されるということをしばしば見出すことがある（例えば誤り処理ルーチンなどはめったに使用されない）．

不幸にしてコードのどの部分が最も重要なのかを前もって決定する方法はない．したがって，できるだけ早期にコードの原型を作成してあい路を決定し，それを最適化することが重要である．

以上が，高水準言語を使用する利益が明らかであると考えられる７つの点である．この利点を完全に利用するためには次の条件が必要である：

1. システム・プログラミングのために十分能力のある高水準言語が必要である．
2. この高水準言語に対してコンパイラが必要である（第８章）．
3. この高水準言語を支える計算機とオペレーティング・システムが必要である．

6.2 高水準言語の特徴

本節では PL/I や ALGOL 68 に現われるような高水準言語の重要な特徴を論じる．これらの特徴のすべてを含んでいるわけではないが，FORTRAN や COBOL，ALGOL，GPL は幾つかの特徴（例えば COBOL のデータ構造，ALGOL のブロック構造と回帰呼出し，GPL のポインタなど）を持っている．これらの特徴が将来の言語の標準的な機能になると確信するし，明日のコンパイラはこれらを処理できなければならない．コンパイラを書くときの難しい問題点は，これまでの文献にしばしば見られたような算術文の解釈に関する事柄ではない．むしろ問題点は，記憶領域の割付けや回帰呼出し，オペレーティング・システムとの密接な関連をコンパイルできるように組込むことがきわめて難しいということである．

PL/I は，それ以前の言語の望ましい特徴を効果的に合併しようという試みである．以下の節においては，次に示したようなシステム・プログラミングに特に関係深い特徴を述べる：

1. 拡大されたデータの型とデータ構造（第6.3節）
   a. 文字列
   b. ビット列
   c. データ構造

2. 記憶域割付けと名前の有効範囲（第6.4節）
   a. 記憶域のクラス
   b. ブロック構造
3. アクセスの柔軟性（第6.5節）
   a. ポインタ
   b. ラベル変数とラベル配列
4. 機能のモジュール化（第6.6節）
   a. （内部および外部）手続き
   b. 回帰呼出し
5. 非同期処理（第6.7節）
   a. 条件処理
   b. シグナル
   c. 多重タスク
6. 拡張性（第6.8節）
   a. コンパイル時のマクロ
   b. データ構造
   c. サブルーチン
7. その他（第6.9節）

6.3 データの型とデータ構造

たいていのプログラム言語において陽に指定して処理することのできるデータの型の種類は限定されている．例えば，FORTRAN においては整数型（固定小数点の数）と実数型（浮動小数点の数）が基本的なデータの型である．複素数や論理変数も，ある種のコンパイラでは使用できる．FORTRAN では文字データを処理できるけれども，データは整数型または実数型として処理しなければならないために，その処理はスマートとはいえず，はなはだやっかいである．アセンブラやローダの設計の項でも見たように，多くのシステム・プログラムの処理において文字やビットのデータの処理は基本的である．さらに，使用されるデータ構造は単純な配列よりもはるかに複雑である．特に，各々の

データ項目は幾つかの異なったデータの型を持つ欄とすることができる．PL/I のビット列や文字列，データ構造はシステム・プログラマの仕事を大変容易にするものである．

```
DECLARE      FIND_SYMBOL    CHARACTER(8);
DECLARE      1 SYMBOL_TABLE(100),
                2 SYMBOL CHARACTER(8),
                2 VALUE FIXED BINARY,
                2 LENGTH BIT(8),
                2 RELOCATION CHARACTER(1);

             ⋮
             /*記号の探索*/
             DO I = 1 TO 100;
             IF SYMBOL_TABLE(I).SYMBOL = FIND_SYMBOL
                THEN GO TO FOUND;
             END;

             ⋮
             /*記号が見付かった*/
FOUND.
```

図 6.1 データの型とデータ・セグメントの例を示す PL/I プログラム・セグメント

図 6.1 は1つの PL/I プログラム・セグメントを示している．データ構造の宣言は，第3章の**図 3.8** に示したアセンブラの記号表と同じ形式で100個の欄を持つ記号表を定義している．**図 6.1** の短い PL/I コードは変数 FIND_SYMBOL[3] の内容と一致する記号の欄を探す遂次探索法を示している．この例を後の節で論じることにする．

6.3.1 文字列

PL/I は3つの基本的な文字列操作を備えている：

1. 長さの演算子：引数によって指定された文字列またはビット列の長さを

3 FIND_SYMBOL は単一の変数を表わす，下線"_"は SYMBOL_TABLE などのように内容をわかりやすく示す名前を付けるために使用される．また，PL/I は書式が自由であり，文字の欄位置が任意である．文はセミコロン"；"で終了し，註釈は /* ちゅうしゃく */ のようにして示される．

求めるもの.

2. 結合の演算子：２つの文字列またはビット列の間に使用されたときは，第１の列に第２の列を続けて第３の列を作る.

3. 部分列の演算子：文字列またはビット列を挿入したり抽出したりするために使用される.

次の例はこれらの機能の使用を示したものである：

```
1.   DECLARE  S    CHARACTER(9);
2.   DECLARE  T    CHARACTER(6) VARYING;
3.            T =  'ABCDE';
4.            S =  T || 'IS A STRING';
5.            T =  SUBSTR(S,6,2);
6.            SUBSTR(S,1,3) = 'DEF';
7.            X =  LENGTH(T);
```

文１は変数Ｓが固定長９文字の文字列であり，文２では変数Ｔは長さが０から６まで変化する文字列であると宣言する．文３はＴに値を割当てる．文４は変数ＳをＴと IS A 'STRING' とを結合した文字列に等しくすることを示す．その結果は左側の９文字 'ABCDEIS A' となる．文５はＴをＳの６字目から長さ２字の部分列に等しくすること，すなわち，文字列 'IS' になることを示す．文６はＳの最初の３字が文字列 'DEF' で置換えられ，その結果値 'DEFDEIS A' を持つことを示す．文７は文字列Ｔの長さ２を変数Ｘに割当てることを示す.

Ｔが可変長の文字列（VARYING）であると宣言することは，記憶領域の割付けと処理のためにコンパイラが難しい処理を行なわなければならず，効率が悪いということを注意する必要がある．コンパイラの計設者はこの種の問題にしばしば直面しなければならない.

6.3.2 ビット列

第6.3.1節で述べた３つの基本的な文字列操作に加えて，ビット列の操作では次の３つが可能である：

1. ビット列間の論理積.
2. ビット列間の論理和.
3. 否定：これは０と１を逆にする単項演算子である.

```
DECLARE     B        BIT(1);
DECLARE     L        BIT(1);
DECLARE     R        BIT(1);

            ⋮

            R = ¬B;          /* R = NOT OF B */
            R = ¬L;          /* R = NOT OF L */
            R = B&L;         /* R = B AND L */
            R = B|L;         /* R = B OR L */
```

ブール演算表

B	L	NOT B	NOT L	B AND L	B OR L
0	0	1	1	0	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	1	1

図 6.2 ビット列演算の例

論理積と論理和の演算は**図 6.2** で定義されている.

6.3.3 データ構造

データ構造とは変数間の関係を宣言する手段であり，PL/I や COBOL などのプログラム言語に備わっている．例えば，**図 6.1** は SYMBOL_TABLE という名の記号表の宣言を示している．記号表の各欄は SYMBOL，VALUE，LENGTH および RELOCATION というラベルが付けられた 4 つの部分（フィールド）から成ると定義されている．これらの部分は異なったデータの型を持ってもよい．例えば，SYMBOL は文字列であり VALUE は整数である．**図 6.1** の宣言は 100 個の欄を持つデータ構造の配列を作る．このような表は表全体として処理することもできるし，1 つの欄を処理することもできるし，または 1 つの欄の 1 つの部分を参照することもできる．例として，SYMBOL_TABLE といったときは表全体を示し，SYMBOL_TABLE(10) としたときは表の 第10欄を参照する．従来の処理は通常，例えば，第10欄の VALUE 部を指すときは SYMBOL_TABLE(10) . VALUE というように，1 つの部分に対して行なわれる.

なお他のプログラム言語ではSYMTBL といったあまり具合の良くない短縮名を使用しなければならないが，PL/I ではSYMBOL_TABLE というように長くて十分に意味をいい表わす名前を使用できる．これらのPL/I が持っている“人間性の要素”についてはここでは述べない．

6.4　記憶域の割付けと名前の有効範囲

多くのプログラム作成の計画においては，プログラムの大きさと必要なデータ領域を小さくすることが重要である．データ領域を節減するには2とおりの方法がある：

1. 必要なときのみデータ領域を確保する．例えば，アセンブラの設計においてはパス1のみで必要なデータ・ベースとパス2のみで必要なデータ・ベースが存在した．
2. 必要な量のみデータ領域をとる．多くの場合には前もって配列や長さの大きさがわからないことがあるし（例えば，大きさは入力データによる場合など），もしコンパイラがその機能を持っているなら領域を作るのを実際に必要になるまで行なわないのが得策である．

PL/I の記憶域のクラスとブロック構造の機能はこれらの制御を行なう機構を与える．

6.4.1　記憶域のクラス

PL/I はプログラマに (1)静的，(2)自動的，(3)被制御 の3つの記憶域のクラスを提供する．

静的記憶域は一定不変でコンパイル時に割付けられる．FORTRAN や他の多くのプログラム言語では静的記憶域のみが使用できる．

自動記憶域は，それを参照する手続きやブロックが実行されているときにのみ割付けられる．例えば，自動変数Aを持つ手続き SUBR があるとすると，SUBR を最初に呼出したときAに対して番地が割当てられる．手続きを終了したときAの領域は他の手続きにおいて使用できるように解放される．SUBR が呼出される度にAに対して別の番地が割当てられることもある．Aは自動記

```
参照番号
   1        BEGIN;      /* 外側ブロック1 */
   2          DECLARE  S,X,N FIXED BINARY;
                :
   3          BEGIN;   /* 内側ブロック2 */
   4            DECLARE  A(N), X FLOAT;
   5            X=S;
   6          END;     /* ブロック2の終り */
                :
   7        END;        /* ブロック1の終り */
```

図 6.3　ブロック構造の例

憶域であり，実行時に自動的に割付けられる．

被制御記憶域は実行時にプログラマが陽に記憶域割付けを制御できるようにするものである．プログラマは実行中必要なときに記憶域のブロックを割付けたり解放したりできる．ポインタについて論じるときにこれの例を示すことにする．

6.4.2　ブロック構造

プログラムの中で記号の宣言が行なわれたとき，この宣言が有効であるプログラムの範囲が明確に定められている．PL/I や ALGOL のようなプログラム言語におけるブロック構造の機能は，変数の宣言の有効範囲を制御する．**図 6.3** はこの機能の使い方を説明している．（図で参照番号はプログラムの一部ではない．それらは説明において行を指すために用いられる.）

図 6.3 の PL/I プログラムの外側のブロックには変数S，XおよびNが整数型（固定小数点2進数）として定義されている．特に指定がないとき PL/I の変数は自動変数であるとみなされる．このプログラムは参照番号3から6までの内側のブロックを持っている．内側のブロックでは，配列Aと変数Xが実数型の浮動小数点であると定義されている．これらの変数は自動記憶域のものであるから，これらは内側のブロックに入ったときにのみ領域を割付けられる．この例は幾つかのことを説明している：

1. このプログラムはXという名前の2つの別の変数を持っているが，ブロック構造によって定義されているように同時には1つだけ使用される．
2. 配列Aは，外側のブロックで既に定義されている整数Nに入っている長さを持つと宣言される．
3. 文X＝Sは，文4にて定義された変数Xを文2にて定義された変数Sの値に等しく設定する．これらの変数は異なったデータの型を持つから，PL/I は自動的に変数Sの値を浮動小数点数に変換する．

6.5 アクセスの柔軟性

システム・プログラミングを効果的に行なうためには，柔軟なデータ参照と制御の移行ができなくてはならない．柔軟なデータ参照とは，リスト構造のような自分で自分のデータ構造を定義したり複雑なデータ編成を処理する能力をいう．柔軟な制御の移行とは，入力データや前の処理結果に基づいて移行先を変更したり設定したりできる機能をいう．柔軟なデータ参照は基底付き記憶域とポインタによって達成される．また，プログラムにおける柔軟なアクセスはラベル変数やラベル配列の機能によって達成される．

6.5.1 ポインタ

ポインタは変数に対する限定詞（qualifier）と考えることができる．例えば基底付き記憶域の場合には，プログラマはAと名付けた記憶域のブロックを割付け，その後同じくAと名付ける記憶域の他のブロックを割付けることができる．ブロックは同じ名前を持っているから，これらを区別するための限定詞が必要である．プログラマが第1のブロックを割付けるとき，この領域を指すようにポインタを設定し，第2のブロックを割付けるときには他のポインタを設定することができる．このようにして，単に適当なポインタを指定することにより2つのブロックを区別することができる．プログラマが陽に領域の割付けを制御する変数は“被制御記憶域”の中にあることを思い出してほしい．

図6.4および**図6.5**はポインタと被制御記憶域の使用法を幾つか説明している．示されたプログラムは**図6.1**に示された記号表処理の例と同様に働くもの

である．前の例では100個の欄を持つ記号表を定義した．実際には前もって必要となる記号欄の個数はわからない．したがって，必要とされる度に1つずつ欄を割付ける方が都合が良い．しかしこのようにすると，これらの欄はもはや連続した領域には置かれないので，各欄の番地を覚えておくことが必要である．これは各欄に“次の”欄の番地を示す部分を追加することによって行なうことができる．**図6.4**に示すポインタとして定義されたNEXT部がこの役割りを果す．ENTERおよびSEARCHという名札を付けられたプログラム・セグメントは，各々新しい欄を作ったり，特定の欄を探索したりする処理を説明している．**図6.5**はこのプログラムによって作られた記号表のデータ・ベースを示している．複雑なデータ・ベースの部分を“結合”したり“連鎖”したり

```
DECLARE       FIND_SYMBOL   CHARACTER(8);
DECLARE       1  SYMBOL_TABLE      BASED(P),
                   2  SYMBOL CHARACTER(8),
                   2  VALUE FIXED BINARY,
                   2  LENGTH BIT(8),
                   2  RELOCATION CHARACTER(1),
                   2  NEXT POINTER;
DECLARE       SYMBOL_TABLE_START POINTER INITIAL (NULL);

                 ⋮

ENTER:     /*    表に新しい欄を作る*/
                 ALLOCATE SYMBOL_TABLE SET(P);
           /*    新しい欄を表の始めに結合する*/
                 P→SYMBOL_TABLE.NEXT = SYMBOL_TABLE_START;
                 SYMBOL_TABLE_START = P;
           /*    新しい欄に記入*/
                 P→SYMBOL_TABLE.SYMBOL = etc.

                 ⋮

           /*    記号の探索*/
SEARCH:          P = SYMBOL_TABLE_START;
LOOP:            IF  P = NULL THEN GO TO NOT_FOUND;
                 IF  P→SYMBOL_TABLE.SYMBOL = FIND_SYMBOL
                        THEN GO TO FOUND;
                 P = P→SYMBOL_TABLE.NEXT;
                 GO TO LOOP;

                 ⋮
```

図 6.4　ポインタと被制御記憶域の例

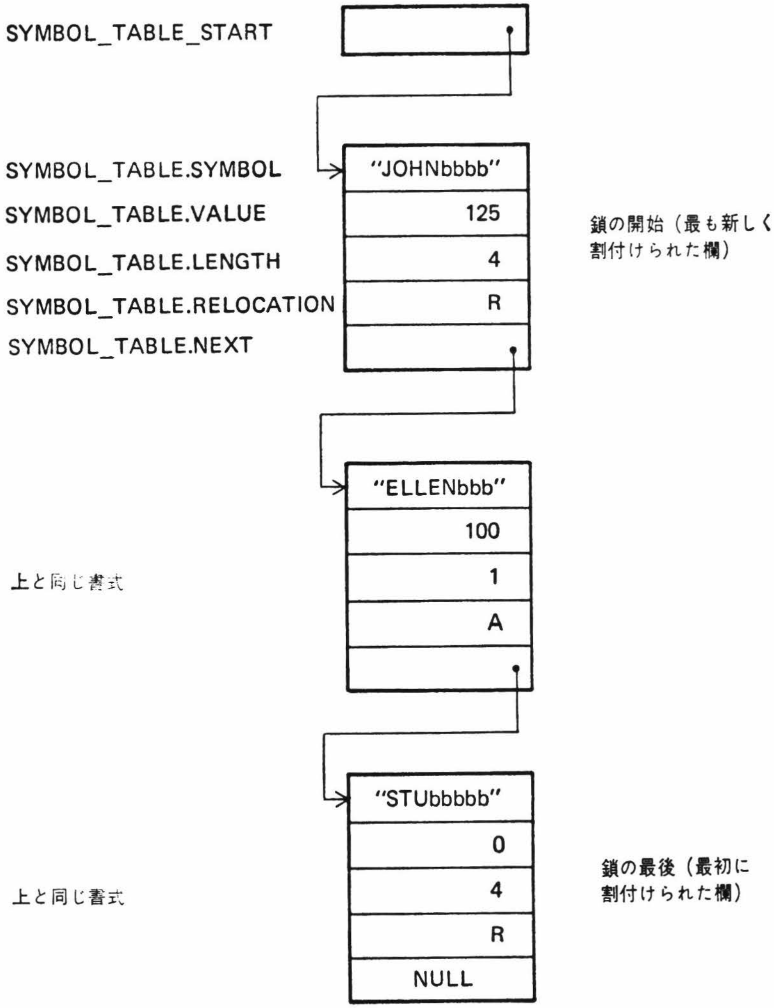


図 6.5　リスト構造にされた記号表

する処理はしばしば**リスト処理**と呼ばれる．コンパイラ組込みの詳細を論じるときに，このような編成の例がたくさん出てくる．

図 6.4 ではデータ構造 SYMBOL_TABLE を持つ記号表の 1 つの欄を定義している．プログラムは新しい欄を作り出し，これらを結合する．プログラム

の他の部分は表に格納された欄を探索する．ALLOCATE 文は，1つの新しい欄としてデータ構造 SYMBOL_TABLE を割付け，その番地をポインタPに格納する．次に，**図 6.5** に示したように，ポインタ SYMBOL_TABLE_START を用いて“第1”欄の番地を指定し，他の欄は NEXT 部を介して“結合”する．欄は新しいものが先に置かれて結合される．しかしこれは単なるプログラム技法の都合である．ポインタの特別な値 NULL は何も指さないということを意味し，この値は鎖の終りを意味するために使用される．

6.5.2 ラベル変数とラベル配列

ポインタがデータの番地を変数として取り扱うことを可能としたように，ラ

```
DECLARE     L(5) LABEL;
DECLARE     PROCESS LABEL;
DECLARE     INST_TYPE FIXED BINARY;
                 :
            GO TO L(INST_TYPE);
                 :
L(1):       /*RR形式の命令*/
            PROCESS=TWO_BYTE;
                 :
L(2):       /*RS形式の命令*/
            PROCESS=FOUR_BYTE;
                 :
L(3):       /*RX形式の命令*/
            PROCESS=FOUR_BYTE;
                 :
            GO TO PROCESS;
TWO_BYTE:
                 :
FOUR_BYTE:
                 .
```

図 6.6 ラベル変数とラベル配列の例

ベル変数は命令の番地を変数として取り扱うことを可能にする．**図6.6**はラベル変数とラベル配列の使用例を示している．記号L(1)，L(2)，……，L(5)はプログラムの５つの文に付けられたラベルを示す．各ラベルはアセンブラのパス２において５種類の命令形式を処理するコードを示すと仮定する．整数 INST_TYPE が命令形式を指定しているとすると，文 GO TO L(INST_TYPE) はL(1)，……，L(5) のどれか１つに制御を移すことになる．INST_TYPE が１から５以外の値を持っていたとすると誤りであることが表示される．

変数 PROCESS はラベル変数である．そしてこの変数を１つのラベルまたは他のラベル変数の内容に等しく設定することができる．命令形式にしたがって，PROCESS はラベル TWO_BYTE，FOUR_BYTE，または SIX_BYTE に等しく設定される．文 GO TO PROCESS が実行されると適当なラベルのところへ制御が移される．ラベル変数を引数とすることおよび内側のブロックから外側のブロックや外部手続きへ制御を移すことが認められているので，ラベル変数を組込むことは一般的な状況では最も難かしいことである．この点については後の章で詳述する．

6.6　機能のモジュール化

システム設計の最も大切な点の１つは，機能単位に分割されたモジュールを設計し，組込み，そしてデバッグする能力である．第５章で論じたローダの機構によって**サブルーチン**とか**手続き**とか呼ばれる個別のプログラム・モジュールを結合することができる．サブルーチンの機能はごく普遍的なものであるけれども，基本 ALGOL や基本 COBOL など多くのプログラム言語はこの機能を備えていない．

6.6.1　手　続　き

PL/I では内部および外部の手続きや関数を作ることができる．内部手続きは単一のプログラム・モジュールの一部として含められるものである．外部手続きは別々にコンパイルされる単独のプログラム・モジュールであり，ローダを用いて他のプログラム・モジュールと結合される．さらに，PL/I の手続き

は複数の入口を持つことができる．この特徴は SINE と COSINE ルーチンなどよく類似したサブルーチンに対してはきわめて有用である．

6.6.2 回帰呼出し

自動記憶域によって可能になった回帰呼出しを用いると利用者は広範囲の問題を自然な形で取り扱うことができる．プログラマが書かなければならないルーチンの多くはコンパイラの構成で見るように回帰的である．またマクロ処理機能の組込みにおいて，回帰呼出しはマクロ定義の中でのマクロ呼出しを可能にするために都合の良い方法であることを見た．

さらに，多くのアルゴリズムは回帰的に記述し，プログラムする方がはるかに都合が良い．例えば，階乗の標準的な数学的定義は次のように表わされる：

$$n! = \begin{cases} 1 & \text{if } n=0 \\ n*(n-1)! & \text{if } n\neq 0 \end{cases}$$

この関数をそのまま組込むと**図 6.7** に示したようになる．

```
N_FACTORIAL:  PROCEDURE (N)     RECURSIVE;
              IF N=0 THEN RETURN(1)
                 ELSE RETURN (N* N_FACTORIAL(N-1));
              END;
```

図 6.7　回帰手続きの例

6.7 非同期処理

たいていのプログラム言語では，単一の **一続きの同期処理**（すなわち，GO TO 文が臨時に実行の流れを変更する以外は，一時に１つの命令を順番に実行すること）を建前としている．システム・プログムミングにおいては次にあげるような正規の手順を中断して処理しなければならない場合，すなわち，同期の取れない処理が数多く発生する．

1. 内部および外部割込みの処理（すなわちプログラムの状態に起因する正規のプログラムの流れの中断と入出力完了や誤りなどの**外部割込み**）．
2. 手続きによって検出され，他の手続きに通知する必要のある結果の無効性などのようなソフトウェア割込み．

3. 同時に2つ以上のタスクの実行を認める多重タスク（多重プログラミング）のオペレーティング・システム．

6.7.1 条件処理

PL/I は高水準プログラム言語として，順序付けられないプログラムの流れを可能にしようとする最初の試みを行なっている．**条件**はプログラマが種々の割込みや特別の状態を処理したり取り扱かったりする便宜を与える．**図6.8** のプログラムは条件の使用例である．

図6.8には，S1，S2 および S3 の3つの条件処理文がある．S1 は PL/I の GET 文の入力動作で入力データの最後に達したならば，制御は直ちにラベル END_PROG の付いた文に移されることを述べている．S2 の条件ハンドラが呼出されると，それ以後のレジスタ**あふれ**によるハードウェアの割込み（ハードウェアのレジスタに納まり切らない大きな計算結果が生じたことを示す）は制御を PUT EDIT（WRITE に対応する）とそれに続く命令に移す．文 S3 の ON OVERFLOW SYSTEM が実行された後にあふれによる割込みが発生すると，あらかじめシステムで定められたプログラムの終結などの，無指定のとき

```
DEMO:       PROCEDURE;
S1:         ON ENDFILE (INPUT) GO TO END_PROG;
              ⋮
S2:         ON OVERFLOW BEGIN;
                  PUT EDIT ('OVERFLOW OCCURRED') (A(18) );
                  GO TO END_PROG;
                  END;
              ⋮
S3:         ON OVERFLOW SYSTEM;
              ⋮
END_PROG:
              ⋮
            END DEMO;
```

図 6.8 条件の例

行なうべき処理が実行される.

6.7.2 シグナル

第6.7.1節に述べたように，条件とその処理は，サブルーチンが呼出された場合でも，それに続く ON 文によって定義し直されるまでは有効である．例えば**図 6.9** は起こり得るプログラムの流れを示している.

手続きAは手続きBまたはCのいずれかを呼出し，それらは手続きDを呼出し，Dは手続きEを呼出す．手続きEにおいては誤りまたは状態を検査する．手続きBとCは共に TRUBL と呼ばれる条件を定義している.

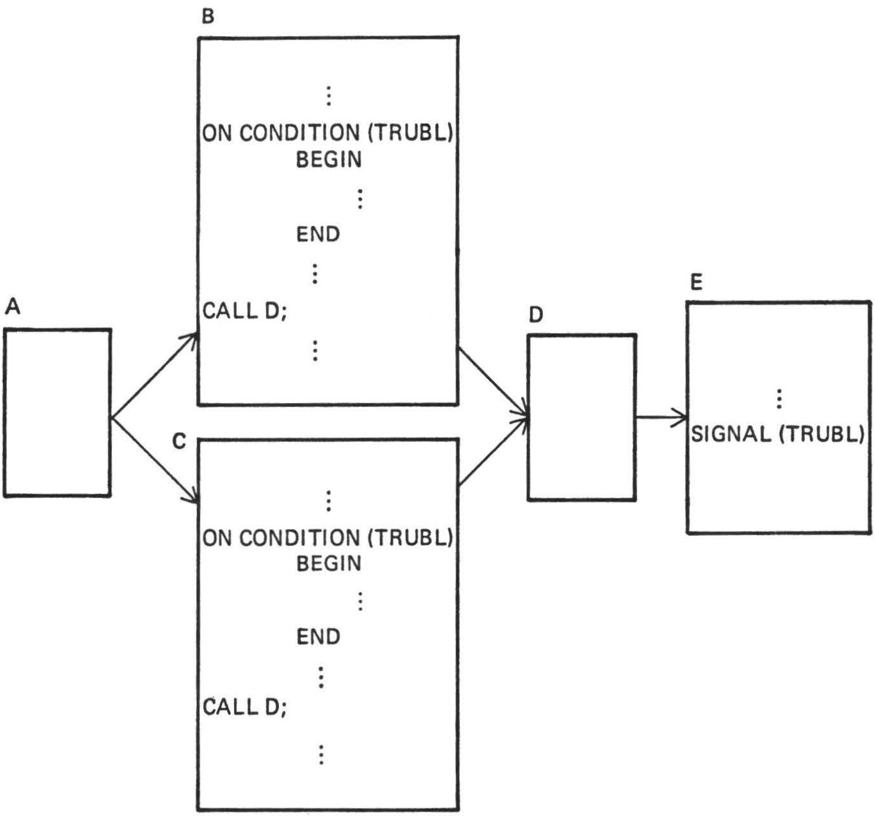


図 6.9 シグナル文の例

もし手続きEがその条件の処理を必要とすると SIGNAL TRUBL を行なう．このとき制御は，どちらの ON 文が有効であるかにしたがって，手続きBまたはCの中に移される．両者の条件ハンドラは，別のものであるかも知れないが，それが支差えのない状態であることを認めると制御を手続きEの直後に返却することができる．

この例でシグナルを用いる利点は，もし TRUBL がBとCとで異なっていたとしても，Dの呼ばれ方にしたがって，シグナルを行なうことにより適当な方の TRUBL が実行されるということである．

6.7.3 多重タスク

PL/I には多重タスク処理の機能があり，多重処理の環境でのプログラムに便宜を与えている．次の手続きはこの機能がどのように働くかを示している：

```
P1:     PROCEDURE;
        :
        CALL P2 EVENT (EP2);
        :
        WAIT (EP2); /*SUSPEND UNTIL EVENT (SP2)*/
        :
        END;
```

前述のプログラムにおいて，WAIT 文は手続き P2 が事象 EP2 を送るまで手続き P1 の実行を中断する．P1 は1つの CPU で，P2 は他の CPU で実行されるような場合を想定している．

6.8 拡張性とコンパイル時のマクロ

データの型，データ構造といった PL/I の種々の機構によって，広い範囲の機能を実現することが可能になる．PL/I コンパイル時のマクロ機能により，プログラマは言語の基本的な機能を用いてより強力なプログラミングの道具を作ることができる．

多くの言語では，アセンブラ言語で宣言されるマクロと同様にして，プログラマが一連の文を定義本体と宣言し，それに名前を付ける機能が備っている．プログラマはこれらの文を，単に定義に名前を付けるだけで，プログラムの中に挿入できる．この特徴は，例えば，同じデータ・ベースを用いたりするような相互に密接な関係のあるプログラムを書く人達にとって便利である．同じ宣言文をすべてのプログラムの中で繰り返さずに，1人のプログラマが1回だけマクロを定義しておくだけで，それ以外のプログラマが作成するプログラムにおいても，コンパイラは適当な宣言文を代入してくれる．

特に，マクロ・プロセッサを適宜使用して PL/I に新しい型の文を追加することも可能である．このようにして PL/I は特定の開発計画のために，より強力で専門化した言語へと拡張することが可能である．

6.9 その他

他の新しい言語と同様に， PL/I は特定の応用に使用される特徴をも持っている．例えば，英国貨幣の単位を示す“スターリング”の宣言は1971年に10進法に切換えられるまで英国の商人に重宝されていた．“浮動小数点”や“複素変数”は技術者にとって都合が良い．

6.10 まとめ

本章で示された特徴は PL/I の機能のすべてを含んでいるわけではなく，通常の言語に見られないもののみを示している．これらの特徴は，将来の，そして，特にシステム・プログラミングのためのプログラム言語において重要な部分となることと確信している．

演 習 問 題[4]

1. 次の記憶域のクラスの各々に対して変数がいつ割付けられ，いつ解放されるのかを説明せよ．

a.　静的記憶域

b.　被制御記憶域

c.　自動記憶域

d.　基底付き記憶域

2.　被制御記憶域と基底付き記憶域の類似点と相違点は何か．

3.　*DOグループ，開始ブロックおよび手続きブロックの相違点は何か．互に置換え可能か．答えを裏付ける例を示せ．

4.　*次の各々は変数の有効範囲にどのような影響を与えるか．どのように実行されるか．どのようにして終了するのか．

a.　関数手続き

b.　開始ブロック

c.　DO グループ

d.　サブルーチン（呼出された手続き）

5.　PL/I プログラマは，変数の有効範囲を制御するのにどのような手段を持っているか．この制御法が言語の有用な特徴である理由は何か．

6.　*値を PL/I の手続きに与える3とおりの方法をあげよ．

7.　*次のプログラムを実際に模擬実行してみよ．1つの文を実行する度に，行番号と共にその文によって変更されたり作り出されたりした変数の名前と値を記録せよ．

```
行番号

 1     WHAT_IS:  PROCEDURE OPTIONS (MAIN);
 2        DCL    (X,Y) FIXED BINARY INITIAL (-2),
                 (W,Z) FIXED BINARY INITIAL (0),
                 M     FIXED BINARY CONTROLLED;
 3     I: PROCEDURE (J);
 4        DCL (J,X) FIXED BINARY;
 5        X=Z;
 6        RETURN (Z*X-J);
 7     END I;
 8     L1: ALLOCATE M;
 9          M=X*Y+W-Z;
10     L2: Z=X+Y;
11          IF Z=0 THEN Z=I(Z);
12          BEGIN:
```

4　*は PL/I の参照マニュアルの使用を要する問題であることを示す．
†は計算機固有の問題であることを示す．

```
          DCL W INITIAL (2) FIXED BINARY;
      L3: Z=I(W);
          IF X=Z THEN GO TO L2;
          IF X < Z THEN GO TO L1;
          W=W+1;
          GO TO L3;
      L1: END;
      X, W=M;
      IF X > 0 THEN GO TO L1;
      Z,Y,W=X;
      END WHAT_IS;
```

8. *この問では，本章では述べることのできなかった高度の機能を用いた PL/I プログラムを読ませようとしており，幾つかの新しい内容が紹介されている．PL/I 参照マニュアルを読むことによって理解できるであろう．

```
1.      BETA : ALPHA : PROC OPTIONS (MAIN) ;
2.                 DECLARE  X FIXED BIN EXTERNAL;
3.                 DCL      Z FLOAT DECIMAL INITIAL (12) ;
4.                 DCL      LABEL_X LABEL ;
5.                 DCL      MESSAGE CHAR (24) VARYING STATIC
                            INITIAL ('NEVER-AGAIN') ;
6.                 X = 5 ;
7.                 ZED = X + Z ;
8,9.               IF ZED < 4 * Z THEN DO ;
10,11.             X = 1 ; LABEL_X = LABEL_1 ;
12.                GO TO LABEL_X ;
13.                END ;
14.                ELSE TRY : BEGIN;
15.                DCL LAMBDA CHAR (0) ;
16.                SUBSTR (MESSAGE, 6, 6) = LAMBDA ;
17.                Z = Z - LENGTH (MESSAGE) ;
18.                UPTO : PROCEDURE ;
19.                DCL Z CHAR (5) INITIAL ('JOHNS') ;
20.                DCL X CHAR (20) VARYING INITIAL (LAMBDA) ;
21.                X = Z || 'TRY-AGAIN' ;
22.                END UPTO ;
23.                NEW = X = Z ;
24.                END TRY ;
25.     LABEL_1 : CALL DELTA ;
26.     GAMMA: PROC ;
27.                DCL X FIXED BINARY EXTERNAL ;
28.                DCL Z FIX BIN (31) CONTROLLED ;
```

```
29.          ALLOCATE Z ;
30.          X = 4 ;
31.          Z = X * * 2 ;
32.          X = X + Z ;
33.          FREE Z ;
34.          END GAMMA ;
35.          UPDATE : AGAIN : PROCEDURE ;
36.          X = X + 4 ;
37.          DELTA : ENTRY ;
38.          Z = X * * 2 ;
39.          Z = Z - 40 ;
40.          END AGAIN ;
41.          CALL UPDATE ;
42.          CALL GAMMA ;
43.          Z = X + SQ (X) ;
44.          SQ : PROC (P) FIX BIN ;
45.          DCL P FIX BIN ;
46.          STR = P + P * 2 ;
47.          RETURN (STR) ;
48.          END ;
49.          END BETA ;
```

上の PL/I プログラムを調べ次の問に答えよ．プログラムの流れを追い，変数X，Yおよび ZED の各文の実行後の値を示せ．**註**：文が“実行”される順序は上に示された番号のとおりではない．開始ブロック，手続きなどの中の制御の流れを見よ．答えは次に示した表の形で示せ．表には幾つかの欄を例として記入してある．

番号	“実行”された文	変数の値		
		X	*Z*	*ZED*
1	1	未定	未定	未定
2	2	″	″	″
3	3	″	12	″
4	4	″	12	″
5	5	″	12	″
6	6	5	12	″
7	7	5	12	17
	,	,	,	,
	,		,	,
	,	,	,	,
	,	,	,	,

“実行”という言葉は宣言文は実際には実行されないので“を付して用い

た．この問題ではインタプリータが上のプログラムを処理しているという意味でこの言葉を用いている．

9. †2進整数をローマ数字に変換するPL/I手続きを作れ．BINARYやDECIMAL と同様な属性でROMAN という属性はない．ローマ数字は文字列で表現されるものとする．数字は文字列の中で右寄せになっている．必要に応じて左には空白を入れること，出力は簡潔な表現でなければならない（例えば，19はXIXによって表わされる）．変換すべき数は正で4000以下である．

ローマ数字は次のとおりとする：

I～1, V～5, X～10, L～50, C～100, D～500, および M～1000

手続きの名前はSPQRで次のようにして呼出されるものとする：

ROMAN=SPQR (BINARY)

10. †ソフトウェアを組込むとき，多くの“ユーティリティ・ルーチン”が必要となる．アセンブラ，ローダおよびコンパイラにおいて，記号表やリテラル表への参照が頻繁に行なわれる．これらの表を作ったり情報を表から取り出したりするためにSYMSTOおよびSYMGETというユーティリティ・ルーチンを使用することができる．PL/IでSYMSTOおよびSYMGETを書け．

a 採点プログラム[5]は新しい記号の値を格納するためにSYMSTOを呼出し，既に格納された記号の値を求めるためにSYMGET を呼出す．引数の1つとして，次のような表が与えられる．その表は先頭から0個または正数個の空白でない欄を持ち，各欄は順番に並べられている．また，次の規約を備けている．

1) 表はこの問題の終りに宣言してあるTABLEというPL/Iの構造体である．

2) 表の欄はSYM要素が‘*bbbbbbbb*’に等しいとき空である．ただし*b*

5 この問題は採点プログラムを用いなくてもできる．この場合，データと主プログラムを用意せねばならない．採点プログラムは学生のプログラムを呼出し，結果の記録を行なう．

は1個の空白を表わす（すなわち，8個の空白からなる文字列）．

3) プログラムが呼出されたとき，表は0個（空白）から100個の空白でない欄を持ち得る．

4) すべての空でない欄は表の先頭からすき間を空けずに置かれている．

5) 欄は綴りの逆の順に並べられている．

6) 表には同じ記号が2回現われることはない．

呼出されたときの表の例：

SYM	VALUE	ATTRIBUTES			
		LENGTH	TYPE	REL_ABS	
TERM	44	16	1	0	第1欄
NEXT1	12	4	5	1	第2欄
FIND2	16	2	4	1	第3欄
ARG	26	8	2	0	第4欄
bbbbbbbb	0	0	0	0	第5欄-空欄
・ ・ 空 欄 ・		空 欄			
bbbbbbbb	0	0	0	0	第100欄-空欄

b. SYMSTO

SYMSTO の入口は次のように呼出される（この文は採点プログラムで実行される．読者のプログラムはNに割付けられる値を返却しなければならない）：

N＝SYMSTO(SYMBOL , TABLE , VALUE , LENGTH , TYPE , MODE) ;

SYMSTO は次のことを行なわねばならない：

1) 次の2進固定小数点（31,0）の値を採点プログラムに返却する：

−1 SYMBOL が既に TABLE の中に存在するとき.
0 TABLE が一杯のとき.
1 SYMBOL が正しく登録できたとき.

2) SYMBOL が TABLE の中になく，TABLE が一杯でないとき TABLE の中に SYMBOL のために欄を追加する．新しい欄は次のように追加されねばならない．使用されている欄の直後の空欄を使用する．直前の欄と比較し，その値が小さければ2つの欄を入換える（すなわち,欄の構造全体を入換える)．この操作を繰り返し，上の6つの拘束条件を満すように新しい欄を正しい位置に置く．

3) SYMBOL のVAL，LENGTH，TYPE および REL_ABS 要素に引数の VALUE，LENGTH，TYPE および MODE の値をそれぞれ設定する．

c. SYMGET

SYMGET は次のように呼出される（採点プログラムはこのようにしてプログラムを呼出す)：

VALUE＝SYMGET(SYMBOL , TABLE , ITEM) ;

SYMGET は次のことを行なわねばならない：

1) SYMBOL が TABLE の中に見付かれば，ITEM の値にしたがって SYMBOL の欄の要素の1つを返却する．

ITEM の値	SYMGET の返却する値
'VALUE'	VAL 要素
'LENG'	LENGTH 要素

'TYPE'　　　　TYPE 要素

'MODE'　　　　REL_ABS 要素

2) 記号が表の中で見付からなければ値−6251を返却する.

3) SYMGET は表 TABLE を修飾してはならない.

入口と引数の宣言

手続きの中で正しく入口と引数を確立するために，プログラムは次の形でなければならない：

```
SYMSTO:    PROCEDURE(SYMBOL,TABLE,VALUE,LENGTH,TYPE,MODE)
              FIXED BIN(31);
DECLARE    SYMBOL     CHAR(8),
           (VALUE,LENGTH,TYPE,MODE) FIXED BINARY(31),
           1  TABLE(100),
              2  SYM CHAR(8),
              2  VAL FIXED BINARY(31),
              2  ATTRIBUTES,
                 3  LENGTH FIXED BINARY(31),
                 3  TYPE FIXED BINARY(31),
                 3  REL_ABS FIXED BINARY(31);
           [ SYMSTO とSYMGET に対する宣言文およびSYMSTO の本体

           [ SYMSTO の本体はRETURN 文で終らねばならない
SYMGET:    ENTRY(SYMBOL,TABLE,ITEM) FIXED BINARY(31);
DECLARE    ITEM CHARACTER(4);
           [ SYMGET の本体 ]
END SYMSTO;
```

11. †後の章で行なうコンパイラの議論において新しいコンパイラには6つの構成要素があることを示す．ここではその内の1つの要素である単語解析プロセッサを PL/I で書いてみる．このプロセッサは演算子と記号とだけを持ち，定数を持たないという単純な単語の構造を処理するようにしてある．単語解析ルーチンの目的は，文字の入力流を読込み，それを文法上の単語（**トークン**と呼ばれる）にまとめることである．したがって次の36字は

FOO : IF ALPHA=BETA THEN GOTO EXIT ;

次の10個のトークンの並びに形成される：

FOO	:	IF	ALPHA	=	BETA	THEN	GOTO	EXIT	;

これから作る PL/I プログラムは，NEXTKN および NEWLINE という名

の2つの入口を持つ．入口 NEWLINE は，採点プログラムが入力カードの流れを模擬するために使用される．採点プログラムは，プログラムに新しい入力の行を与えようとするとき NEWLINE を呼出す．入口 NEXTKN は，採点プログラムによって2つのデータ・ベース SYMTAB および OPTAB の現在の内容を伝え，解析ルーチンを呼出すために使用される．採点プログラムは1行の処理中に1回の呼出しで連続したトークンを1つずつ求めるために NEXTKN を何回か呼出す．

NEWLINE は次のように呼出される：

```
CALL NEWLINE(PL/I の文字列)；
```

呼出されたとき読者のプログラムはこの引数を，これから処理する新しい入力の行として STATIC の文字列変数に保存しなければならない（なぜか）．渡される行は長さが変化するが次の関係を満たしている：

$$1 \leq \mathrm{LENGTH(NEW_LINE)} \leq 80$$

入口 NEXTKN は次のように呼出される：

```
CALL NEXTKN(SYMTAB,OPTAB,UNIFSYM)；
```

呼出されたとき読出しプログラムは次の処理を行なわねばならない：

a. 入力の行から次のトークンを文字列として分離する．
b. 引数 OPTAB および SYMTAB を探索して，トークンが演算子か記号かを決定する．両方の表に見付からないときは新しい記号であると仮定する．
c. 新しい記号のときは，SYMTAB の中で最初に出てきた空白の欄に格納する．SYMTAB に既に入っていればその位置を記録する．演算子の場合には OPTAB の中の位置を記録する．
d. 復帰する前に次の2つの値を渡された引数 UNIFSYM に格納する．
   1) 構造の要素 UNIFSYM.TYPE に1文字を設定する．

      ‘O’ トークンが演算子のとき．
      ‘I’ トークンが記号のとき．
      ‘L’ トークンがリテラルのとき．

SYMTAB の中で記号 'I' またはリテラル 'L' は次のように区別する．リテラルは数字（整数）だけでできている文字列であが，'記号は少なくとも1個の非数字文字を持たねばならない'.

2) 構造の要素 UNIFSYM.INDEX に対応した表の中のトークンの位置の値を設定する．

e. RETURN を実行して採点プログラムに復帰する．(TYPE, INDEX) の対はトークンの不偏記号表現であり，それを一意的に識別することを注意してほしい．

仕　様

入口 NEWLINE については上で十分に述べたが，NEXTKN の詳細については多少補足が必要である．

1) **表**：NEXTKN に関係する2つのデータ・ベースは OPTAB と SYMTAB である．両者共に文字列の単純な配列である．文字列は演算子，記号，またはリテラルの綴りである．これらの表の PL/Iの宣言は次のとおりである：

```
DECLARE  SYMTAB(*)      CHAR(20),
         OPTAB(*)       CHAR(8);
```

これら2つの配列は可変次元であることが宣言の中で "*" により示されている．しかし次の条件を満たすものとする．

1≤DIM(EITHER_TAB, 1)≤16

2) **トークンの分離**：呼出されたとき，NEXTKN は入力の行を，最後に見付けたトークンの後にくる最初の空白でない文字から開始して走査する．トークンを分離するためにはトークンの終りを見付けるまで走査を続けなければならない．次の規則がトークンの終りを規定する．

a. 空白はトークンを終結させる．

b. 行の終りはトークンを終結させる．行は最大80文字まで．長さが可変であることを思い出せ．

c. 次の14個の中断文字はトークンを終結させる．また，その文字自身は別のトークンである {：，；，+，−，←，→，>，|（たて棒），‖（2本棒），**，(，)，，（コンマ），<}

3) **返却値**：NEXTKN はトークンを3種に分類し，構造の要素 UNIFSYM . TYPE に，型に対応して上述の CHAR(1)の文字を設定する．次に構造の要素 UNIFSYM . INDEX にトークンの“位置”の値を設定する．“位置”は1から16の間の FIXED BIN(31)の値で，表の中でのトークンの位置を示す指標（添字）である．新しく見付かった記号は SYMTAB に格納されることを思い出せ．

例

採点プログラムは最初解析ルーチンを次のように呼出して，入力の行を引渡す：

```
CALL NEWLINE('IHTFP : TUITION=2500 ‖ 'IS TOO DARN
MUCH ';' ) ;
```

この行は8つのトークンを持っている．次の表が既に与えられていると仮定する：

SYMTAB(7)

1	TUITION
2	2150
3	USAGNP
4	NOV4
5	'　　'
6	'　　'
7	'　　'

OPTAB(9)

1	:
2	←
3	DO
4	=
5	‖
6	END
7	'
8	BEGIN
9	;

6 NEXTKN および NEWLINE の両方の宣言は手続き全体に適用される．入口 NEWLIEN は単に読者のプログラムを外から実行できるように，入口 NEXTKN で開始せず NEWLINE から開始する方法である．関連した入口のところに宣言を書くことで，プログラムは論理をより明確にして構成でき，信頼性と読解性を与える．

それ以後の NEXTKN への呼出しは次の値を返却する.

CALL #	(TYPE,INDEX)	
1	('I',5)	IHTFP は登録された
2	('O',1)	
3	('I',1)	
4	('O',4)	
5	('L',6)	2500 は登録された
6	('O',5)	
7	('I',7)	'IS TOO DARN MUCH' は登録された
8	('O',9)	

採点プログラムとの結合[5]

正しく入口とパラメータを確立するために，プログラムは次の形でなければならない:[6]

```
第2欄
↓
NEXTKN:PROC(SYMTAB,OPTAB,UNIFSYM);
          DCL SYMTAB(*)    CHAR(20),
               OPTAB(*)     CHAR(8),
               1 UNIFSYM,
                 2 TYPE CHAR(1),
                 2 INDEX FIXED BIN(31);

          〔NEXTKN の宣言とコード〕
第2欄
↓
NEWLINE:ENTRY(NEW_LINE);
          DCL NEW_LINE CHAR(80) VAR;

          〔NEWLINE の宣言とコード〕
END NEXTKN;
```

12. †a. IBM 360 アセンブラ語のサブセットを定義せよ.
    b. PL/I でそのアセンブラを書け.

13. †a. マクロ言語を定義せよ.
    b. PL/I でそのマクロ・プロセッサを書け.

第 7 章

形式的体系とプログラム言語：序説

本章の目的は次に示すとおりである：

1. 形式的体系と文法の非形式的な紹介を行なうこと.
2. プログラム言語の領域で行なわれている理論的研究と実際問題との関連を示すこと.
3. 文献などで一般的に使用されている形式的体系と用語を示すこと.
4. コンパイル技法に関した理論的な道具を与えること.
5. 言語を記述する2つの方法，Backus Naur Form と正準体系を述べること.
6. この領域で行なわれている研究の例を示し，プログラム言語と形式的体系の理論に関する研究課題を幾つか示すこと.

本章では複雑多岐にわたる対象を簡単に一覧する．より完全な議論は第10章の参考文献に示されている.

第7．5節と問題1にあげた Backus Naur Form による記述以外，本章では実用的なシステム・プログラムの技法に触れない.

7.1 プログラム言語における形式的体系の使用

形式的体系とは実行されない計算または兵たん術である．形式的体系はアルファベット，公理と呼ばれる単語，および，書換え規則と呼ばれる有限個の関係から成っている．形式的体系の例としては，集合論，ブール代数，命題・述語演算，Post の体系，Euclid の平面幾何学，Buckus Naur Form および Peano の算法などがある．形式的体系は，記号に何らの意味付けもなされていないと

いう意味で，それが**計算**されることがない．上に示した体系に対しては記号の標準的非形式的な解釈が与えられているが，他の解釈もそれらの体系自身に関する限り一般に有効である．

非形式的で直観的な概念のモデルを形成するために，形式的体系を一般的に構成する．正式のモデルを用いるとこれに数学的な考察を加えることができ，モデルが適切であれば，モデルが示す概念よりもはるかに多くの結果を導き出すことができる．形式的体系はプログラム言語の設計，組込み，および研究に対して一層重要になりつつある．特に，各種の形式的体系は構文の記述や，直構文コンパイル，コンパイラの証明，複雑さの研究，および言語の解析に使用されている．

7.1.1　言語の記述

形式的体系はプログラム言語の形式（**構文**）を定義するのに使用される．このような定義は言語の利用者および作成者の双方に重要である．利用者は（参照のために）言語の明確な記述を必要とする．作成者は概念の伝達と保存の問題に直面する．もし同じ言語が別の機械に組込まれる（移し換え）とすると利用者に対しては，外見的に可能な範囲で変化がないように，言語に許された書き方が明確に定義されていなければならない．さらに，作成者は保守に関与しなければならない．コンパイラの利用者と保守者は，共に，対応する言語に許容される書き方の正確な記述を必要とする．

7.1.2　直構文コンパイラ

直構文コンパイラは原始言語入力を**構文解析**（そのストリングを生成した規則を認識すること）するために原始言語の構文規則を納めたデータ・ベースを使用する．形式的体系はそのためのデータ・ベースとして使用される．プログラム言語や機械の種類が増加したので，研究者はコンパイラの自動的な生成を研究している．その研究法は入力原始言語 L_{SOURCE} と出力言語 L_{OBJECT} の形式的な定義を得ることであった．コンパイラ・ジェネレータの出力は $L_{\text{SOURCE}} \rightarrow L_{\text{OBJECT}}$ を行なう翻訳ルーチンTである．

言語処理プログラムの試験を行なうためのテスト・プログラムを生成するこ

とは，自動的なコンパイラ作成と同様の問題である．もし原始言語が形式的に定義されていると，次節に述べる生成法はテスト・プログラムを自動的に作成する手段となる．計算機は人間の試験者よりも物知りであり得るから，この方法はソフトウェアの試験に対して有効である．

7.1.3 構造の複雑さの研究

形式的体系はプログラム言語とそのコンパイラの複雑さの解明に使用できる．コンパイラの作成者と言語の設計者は，言語にどのような特徴を含めるとコンパイラの認識段階が過度に複雑になるか知ることを望んでいる．コンパイラ作成者はコンパイラの効率を評価する基準をも求めている．作成者は翻訳処理に期待し得る（ステップ数で表現した）理論的な最適効率を知りたい．これがわかるとコンパイラの組込みを終えたときに，それの効率を理論的な限界と比較できるであろう．そしてそれが許容範囲内であればこれ以上改善することを止める．しかしもし効率が理論的な限界の半分以下であれば改良を試みるべきであろう．

Shannonの情報理論に関連する研究にも同様のことが見出される．Shannonは情報の測度を決定した．そしてそれを情報の符号化に応用した．得られた定理は情報符号化と伝送の限界を与える．情報伝送用の符号化技術を発明したBell 電話研究所の研究者は，彼等の方法の効率を Shannon の限界を規準として比較し判断することができた． Shannon の定理は符号化の方法を決定するものではない．それは単にどの程度まで効率良くできるかという尺度を与えるにすぎない．同様に，複雑さの考察だけでは，コンパイラの作成者はどのような方法を用いたらよいかはわからず，単により優れた方法が存在する可能性があるか否かということがわかる．

7.1.4 構造解析

形式的体系を用いてプログラムの等価性と有効性の証明が試みられている．プログラムの等価性を証明しようとする研究は，プログラム全体の最適化を期待して始められた．もし２つの別個のプログラムの等価性を判定するアルゴリズムが存在すれば，速いプログラムが遅いものの代りに使用できるであろう．

形式的な定理は各種の言語を解析したり比較したりするための骨組みを提供する．また次のような問に答えるのに役立つ：

1．基本的な言語の特徴は何か．
2．言語の構成法としてどのようなものが存在するか．特徴をどのようにして組合せ，新しい構造が作れるか．
3．どのような範ちゅうの問題がこの言語でプログラム可能か．
4．プログラムを書く経費や難しさはどれ位か．

これらの基本的な質問は形式化によって解明することができる．その答えはハードウェアの設計の分野においても同様に興味のあるものである．理想的な計算機は，言語の基本機能に対応する処理を効率良く実行できなくてはならない．

7.2　形式的定義

7.2.1　形式化の方法

形式化を進める前に，言語を形式的に定義して問題の解析を試み，定義に対する直観的な基本事項を理解しておくことは有用である．言語は明確に定義された構造と意味を持った文または公式——記号の**ストリング**——の集合と考えることができる．言語の有効な構成を規定する規則は**構文**である．言語の構文はその形式を定義する．例えば，$x+2$は表現であるが$x\ 2+$は違うというときは，代数の構文のことを述べている．記号や式に対する意味の割付けまたは**解釈**は，言語の**意味論**（semantics）である．例えば，$x+2$の値はxの値と２の和であるとか，$2\cdot x=x+x$は真であるとかいうときには，代数の通常の意味論を述べているのである．

もしすべての言語が有限個の正しい文や式から成っていれば，構文の定義は何ら問題とならないであろう．このときは単にすべての正しい文を一覧表にするだけで十分である．記号のストリングはそれが一覧表にのっているときにのみ文となり得る．定義の問題は，実用化されている言語はたいていの場合無数の（またはきわめて多くの）正しい文を持つことに在る．無限の言語に対して

すべての正しいストリングの一覧表を格納することは不可能である．しかし，すべての文を発生させることは際限がないとしても，必要になったとき一覧表の文を生成することができれば，文の一覧表を格納しておくことが不用となる．順次に正しいストリングを作り出すアルゴリズムが存在すれば，表を生成することができ，任意のストリングは，生成された表の中にそれが存在すれば，その言語に属することがわかる．もしそれが正しいストリングであれば（時間がかかるかもしれないが）有限の時間経てば生成される．このような一覧表を作るアルゴリズムは言語の**生成的定義**と呼ばれる．

生成のアルゴリズムが，新しく文を生成するときそれ以前の文より短くない（少なくとも同数の記号を持っている）というような順番で文を生成するならば，与えられたストリングが言語に含まれるか否かを決定することが可能である．調べているストリングが未だ生成されていないときに，それより長い文が生成され始めたとすれば，このストリングは言語に含まれ得ないということが決定できる．この型のアルゴリズムによると，有限回の手順を経て，1つのストリングが正しい文であるかどうかを決定することができる．すべてのストリングに対してこのような決定が有限時間内に行なえるならば，この言語は**決定可**（decidable）と呼ばれる．

別の型のアルゴリズムも言語を定義するために使用できる．この第2の方法では調べようとするストリングがアルゴリズムの入力として加えられる．アルゴリズムは入力を解析し，必要な計算を遂行し，“ストリングが正しい”か“正しくない”かを答える．これは**解析的定義**と呼ばれる．解析的定義の言語は，アナライザがすべての入力ストリングに対して有限回の手順の後でつねに停止すれば決定可能である．しかし形式的な解析的定義を作り出すことはしばしば非常に困難である．本章では主として生成的定義を扱うことにする．

英語は余りにも漠然としており，あいまいな定義になってしまうので，形式的に言語を定義するには不適である．言語の定義ができるような形式化の手段を開発することが必要である．この定義用言語が**構文用超言語**（syntactic meta-language）である．1つの言語がある言語（自分自身または他の言語）を述べるために使用されるとき，前者を**超言語**，後者を**目的言語**（object language），と呼ぶ．形式的体系は超言語である．目的言語の記号は**終端記号**

(terminal symbol) と呼ばれる．目的言語のストリングを示す超言語の記号は**非終端記号** (nonterminal symbol) と呼ばれる．超言語を形式的に定義するには超超言語が必要である．したがって超言語は直観的に意味が明らかであることが望まれる．

定義の過程の第1歩は**議論の万有性**を確立することである．すなわち，議論の対象物を定めておくことが必要である．最も基本的な対象は記号である．記号は結合されてストリングとなり，ストリングは言語に含まれたり含まれなかったりする．

定義1： **アルファベット**Tは記号(“終端記号”)の有限の集合である．式(ストリングや文とも呼ばれる)は記号を結合したものである．

アルファベットTでできる有限のストリングの集合を考え，これを T^* と表わすことにする．任意の集合Uに対して U^* は，集合Uの要素のすべての可能な有限の結合を表わす．ストリングを表わすためにギリシア文字を使用する．“null”または“空”のストリング(すなわち何も要素を含まないストリング)を表わすには普通 λ を用いる．

通常の言語はアルファベットの可能なストリングのすべては含まない．言語においては一部のストリングのみが正しい表現である．
すなわち，

定義2： **言語**LはアルファベットTの有限な結合の集合の部分集合である．これを $L \subset T^*$ と記す．

7.2.2　形式的定義の発展

さて例として英語の構文を考えてみよう．英語は単なる単語のグループの集合ではなく，単語を結合するための基礎的な構造が存在する．

“The student studies hard.”

という文が与えられたとすると，**図 7.1** に示されたような文の構造を組立てることができる．特に，名詞句と動詞句を分割し，これらの句を個々の単語に分けて解析を完成することができる．すべての文は構造を持っているから，少

ない手順でこの構造を生成し，複雑な文章を作り上げることが可能である．構造を**構文の樹**を用いて図式的に表わすことができる．樹の各節点からの枝は論理的な分割を示している．

例えば，“文”から生成を開始して，これを名詞句と動詞句の対で置換える

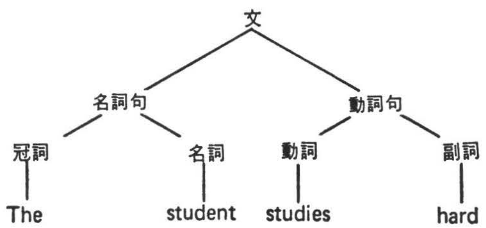


図 7.1 “The student studies hard”に対する構文の樹

ことにより文がとり得る形式の１つを構成することができる．この生成を

文 → 名詞句　動詞句

と書くことにすると何かと便利である．

文脈から名詞句は言語学でいう“名詞句”に属するものを表わすことは明らかである．**図 7.1** でストリング“名詞句”とストリング“The”とを区別しなければならない．“名詞句”はストリング“冠詞”およびストリング“The”と置換えができ，“The”はもう置換えることができない．言語学的な品詞記号の両端を超かっこ“〈”および“〉”でくくって品詞の名前と言語の単語とを区別する．構造の法則の最初は次のようになる：

(1)　〈文〉 → 〈名詞句〉〈動詞句〉

(2)　〈名詞句〉 → 〈冠詞〉〈名詞〉

同様にして動詞句は副詞で修飾されることのある動詞から成っている：

(3)　〈動詞句〉 → 〈動詞〉

(4)　〈動詞句〉 → 〈動詞〉〈副詞〉

これらの構造変換則——書換え則——が定められたことにより〈動詞句〉は１つの副詞を持ってもよいが〈名詞句〉は１つの形容詞を持たなければならない．

最後の段階は〈冠詞〉，〈副詞〉，〈名詞〉および〈動詞〉という品詞記号を置換えることのできるすべての終端記号をあげることである・

(5)　〈冠詞〉 → The

(6)　〈名詞〉 → student

(7)　〈動詞〉 → studies

(8)　〈副詞〉 → hard

(9)　〈副詞〉 → slowly

これらの書換え則を用いると，与えられている構造の言語学の品詞記号を置換えることによって文章を作ることができる．**図 7.2** は "The student studies hard." の生成を示す．最後の段階を変更して

〈副詞〉 → slowly

を用いると，代りに "The student studies slowly" を生成することが可能となる．構造書換え則，または**変換** (transformations)，は生成される言語の形式を決定する．上の例では，言語は英語のきわめて小さな部分集合である．

文章を生成するためのアルゴリズムは上のような書換え則の体系で構成される．記号を単語から音素に変更し，より複雑な構造変換を認めると上と同様にしてより多くの英語の構文を記述することができる．同様にこの方法は多くのプログラム言語の特徴を取り扱うことができる．しかしながら，もっと記述能力を要する特徴が存在することから，他のより識別力のある言語定義の手段を求める必要が生じる．

ステップ	構　造	適用則
a)	〈文〉	(1)
b)	〈名詞句〉〈動詞句〉	(2)
c)	〈冠詞〉〈名詞〉〈動詞句〉	(3)
d)	〈冠詞〉〈名詞〉〈動詞〉〈副詞〉	(4)
e)	The 〈名詞〉〈動詞〉〈副詞〉	(5)
f)	The student 〈動詞〉〈副詞〉	(6)
g)	The student studies 〈副詞〉	(7)
h)	The student studies hard	(8)

図 7.2 導出のステップ

7.3 形式的文法

上の例は，言語に含まれるストリングを生成する過程を定式化する方法である．形式的な定義を考える前に文の構造を解析することは有用である．上では2種の記号を用いた．1つは言語学的な品詞（形式的な生成過程の中間段階として使用される文法の要素）を示すための**かっこにくくられた記号**であり，他はローマ字で綴られ，最終的に文がこれを用いて生成される記号である．生成が終了したとき1つの集合に属する記号のみが文に含まれ，他方の集合に属する記号は中間段階でのみ現われるので，これらはそれぞれ**終端記号**および**非終端記号**と呼ばれる．非終端記号の1つである**開始記号**（例では〈文〉）はこの記号によって生成過程が開始されるものとして認識される．

定義3： **終端記号**はアルファベットTの記号である．**非終端記号**はTの中にない記号の集合Nであり，生成過程の中間段階を示す．**開始記号**は言語のすべてのストリングがこの記号から派生するような特別の非終端記号である．

生成過程自身は各段階に任意の書換え規則，または生成則，を適用することである．この過程はストリングを新しいストリングに変換する．この過程は適応できる生成則がないとき，または，ストリングが終端記号のみから成っているとき停止する．

定義4： **生成**（production）[1] とは，変換を行なうストリングのサブストリング（全部のこともある）に左辺のパターンが一致したとき，その部分を右辺によって置換えることを示すストリングの変換規則である．

ストリングの任意のサブストリングを適用可能な生成によって置換えること

1　生成(則）という言葉は，同様な規則を用いて言語を最初に定義した数学者 Emil Postによっている．ここで述べている形式的文法は，生成における記号の使用法に多少の制限があるほかは，基本的に Post の正準体形と同じである．

ができる．この場合生成の左辺と一致したストリングの部分のみを変化させることが必要である．結果的には生成はサブストリングをすべて置換えることもあるし，一部のサブストリングの記号を単に並べ直すだけのこともある．

定義5：　形式的文法GはG＝(N, T, Σ, P) にて表わされる．ただし，

(1)　Nは非終端記号の集合．

(2)　Tは終端記号の集合．

(3)　Σは開始記号$\Sigma \ni N$.

(4)　Pは生成 $\alpha \rightarrow \beta$ の集合である．ただし，

$\alpha, \beta \in (N \cup T)^*$

$\alpha \neq \lambda$（すなわち，αは空でないこと．）

(5)　N∩T は空である．

条件(5)は，終端記号と非終端記号とはつねに区別可能であることを保証する．

7.3.1 形式的文法の例

形式的文法の概念を明らかにするために2つの例を考察する．非終端記号は大文字のローマ字（A, B, C）およびΣにて表わし，終端記号は小文字のローマ字（a, b, c）にて表わす．

例 1： N＝{A, B, Σ}　T＝{a, b}

P＝{Σ→AB　(1)

A→aA　(2)

A→a　(3)

B→Bb　(4)

B→b}　(5)

生成(2)では非終端記号Aが規則の両辺に現われることに注意．(2)は，ストリングのクラスAが“a”で前置されることを示す．クラスAの構造を指している生成は，(2)のほかには(3)だけであるから，Aは“a”の有限のストリング a, aa, aaa, aaaa, ………から成ることが容易にわかる．同様にして，クラスBは“b”の有限のストリング b, bb, bbb, bbbb, ………から成る．文法によって生成される言語は“a”のストリングに“b”のストリングを続けることによっ

て作られるすべてのストリングから成る.

例　2: $N=\{A, \Sigma\}$　　$T=\{a, b\}$

$P=\{\Sigma\rightarrow A$

$A\rightarrow aAb$

$A\rightarrow ab\}$

終端のアルファベットは例1と2とで同様であることから，両方の言語は“a”と“b”とを含むストリングの集合の部分集合である．しかし，これらの間には違いがある．例1の言語においては任意個数の“a”が任意個数の“b”に先行する．しかし，例2では“a”が生成される度に“b”も生成される．したがって，例2の正しい文は“a”のストリングに同じ長さの“b”のストリングが続いたストリングから成る.

7.3.2 文の導出

前節までは形式的文法が言語の生成的定義を行なうことを示した．しかし，ストリングを生成する過程の定式化した定義を行なっていない.

定義6：$\mu=\sigma\alpha\tau$　$\gamma=\sigma\beta\tau$　かつ

$\alpha\rightarrow\beta\in P$

のときに限り，文法Gにおいてストリングγはストリングμから**直接的に導出される**という（$\mu\Rightarrow\gamma$と書く).

ただし，αとτは（空も含めて）任意のストリングとする.

上の例1を参照して，$\mu=aABb$ および $\gamma=aaBb$ とすると，$\sigma=a$，$\alpha=A$，$\tau=Bb$，$\beta=a$ および生成 $A\rightarrow a$ のとき，$aABb\Rightarrow aaBb$ は直接的導出である．次に文法における証明を定義する.

定義7：ストリングγは文法Gにおいて次のときに限ってストリングμから**導出**される（$\mu\Rightarrow^*\gamma$ と書く）という.

ストリング $\omega_0, \omega_1, \ldots\ldots\ldots\omega_{n-1}, \omega_n$　$n\geqq0$ が存在し，

$\mu=\omega_0$

$\gamma=\omega_n$

$0\leq i<n$　　なる i に対して $\omega_i\Rightarrow\omega_i+1$

（すなわち，$\mu=\omega_0\Rightarrow\omega_1\Rightarrow\ldots\ldots\Rightarrow\omega_{n-1}\Rightarrow\omega_n=\gamma$）

ただし，すべての i について，$\omega_i\in(N\cup T)^*-\lambda$

$\{\omega_i\}$ のリストをＧにおける γ の**証明**という．

これは導出過程において，新しいストリングはＧの生成の適用により，以前に導出されたストリングから生成されなければならないということを定式化して述べたものである．最後の条件は，ストリングが**空ストリング** λ（すなわち．記号を持たないストリング）から導出されることを除外するものである．例１の文法における典型的な導出は　$\Sigma\Rightarrow AB\Rightarrow aAB\Rightarrow aABb\Rightarrow aaBb\Rightarrow aabb$，したがって　$\Sigma\Rightarrow^* aabb$（同様に $aAB\Rightarrow^* aaBb$など）である．

7.3.3　文形式と文

上述の定義は生成過程を定式的に記述したものである．ここで，適用可能な導出の集合のいずれが言語のストリングを終結させるか示すことが必要である．

定義８：　**文形式**とは開始記号 Σ から導出される任意のストリングをいう．

例１の文法において文形式の例は aAB や aabb である．文形式または“文のような”形式は最終的なストリングに非終端記号を持った式を含む．

定義９：　**文**とは終端記号のみを含む文形式をいう．

したがって，aabb は文であるが aAB は文ではない．

定義10：　文法Ｇによって定義される言語Ｌ（Ｌ(Ｇ) と書く）はＧの Σ から導出できる文の集合である．

$L(G)=\{\omega\in T^* \mid \Sigma\Rightarrow^*\omega\}$

Ｌの１つの式に対して，導出の各段階において最も左側の非終端記号を置換えるのに，２つ以上の別の証明が存在するとき，Ｌは**あいまい**であるという．

7.4　言語の階層構造

生成の定義は広範な種類のストリング変換を作り出す．生成の形式に，ある種の制限を課すと形式的言語のクラスのサブクラス，例えば，線型文法や生成正則言語など，を生成する文法ができる．Noam Chomsky は，そのような制限にしたがって，言語を分類する4つの言語体系を構成した．

最も一般的な型の文法は生成に何ら制限を課さないものである．特に，記号を抹消（消去）する生成が認められる．このことは中間のストリングが伸びたり縮んだりすることを許す．消去の生成例は aAB→aB であり，文脈 aAB からAが消去されている．制限のない（今定義したような）文法を**0型文法**と呼ぶ．

各生成の右辺が少なくとも左辺と同じ個数の記号を持たなければならないという最も単純な制限を付けると，他のクラスの言語よりも能力の小さな言語が生成される．この制限を持った文法を**1型文法**，または，$\alpha \Rightarrow \beta$ は長さ$(\beta) \geq$ 長さ(α) のときのみ可能であるから**非縮小文脈規定文法**という．

1型文法の生成の例は：

bB→Bb　（記号の交換）

$\sigma\alpha\tau \rightarrow \sigma\beta\tau$（ただし，長さ$(\beta) \geq$ 長さ(α) とする）

定義11： ストリングの**長さ**とはストリングの中の記号の個数をいう．α が1つの記号から成っているとき 長さ$(\alpha)=1$．長さ$(\lambda)=0$（空ストリング）．ストリング $\alpha\beta$ に対して 長さ$(\alpha\beta)=$長さ$(\alpha)+$長さ(β)．

0型および1型文法においてαは任意のストリングでよい．“文脈規定”とはある種の生成は文脈を識別することを示す．例えば，$\sigma\alpha\tau \rightarrow \sigma\beta\tau$ の場合，変換 $\alpha \rightarrow \beta$ は文脈 $\sigma\alpha\tau$ のときに限って行なわれる．1型文法の例は；

例 3： $N=\{A, B, \Sigma\}$　　$T=\{a, b, c\}$

$P=\{\Sigma \rightarrow Abc$

$Ab \rightarrow aAbB$

$Bb \rightarrow bB$

$Bc \rightarrow bcc$

$A \rightarrow a\}$

この文法は $a^n b^n c^n$　$n \geqq 1$　の型のストリングを生成する．

生成の左辺が1つの非終端記号に限定される場合にはその適用は記号が現われる文脈に依存しない．この制限（および右辺が空でないストリングであること）を持つ文法を**2型文法**，**文脈自由文法**または**単純句構造文法**と呼ぶ．後者の名前は文 "The student studies hard" を生成するのに使用した方法と類似していることに由来する．上述の文法は単一記号の制限を満たし，すべての非終端記号は1つの単語または句に展開される．例えば，〈文〉は〈名詞句〉と〈動詞句〉の結合となる．さらに2つの文脈自由文法が上述の例1と2に出てきた．**限定文脈言語**と呼ばれる文脈自由言語のサブクラスは実用的なコンパイリングにおいて重要となってきた．

生成に対する第3の制限は各段階で作り出す終端記号および非終端記号の個数を制限するものである．高々1つの非終端記号が生成の右辺と左辺の双方に用いられるとき生成は**線型**であるといわれる．非終端記号がすべての生成において右辺において右端に現われるとき，生成は**右線型生成**であるという．また，非終端記号が左端に現われるとき**左線型生成**であるという．各生成が右（左）線型であるとき文法は右（左）線型であるという．右線型文法または左線型文法によって生成することができる言語は**正則**であるという．

上述の制限の各々はそれ以前に述べた制限を含む．これらの型は**図 7.3** にまとめたような階層構造を形成する．なお，3型の文法は例2で定義した言語を生成できない．したがって，3型は2型の真の部分集合である．同様に2型の文法は例3の言語を生成できない．したがって，2型は1型の部分集合である．最後に，1型は0型の真の部分集合である．それを生成できる文法の型にしたがって言語を分類する．i 型の言語は i 型の文法（しかし $i+1$ 型（$i=0, 1, 2$）の文法では生成できない）で生成できる言語である．

言語はそれを受理する機械(例えば翻訳プログラムやインタプリータ)によって定義することもできる．これらの一般的な型の言語の各々に対して対応する抽象的な機械が存在する．例えば，正則言語は有限状態の機械によって認識できる言語である．一方0型言語は Turing の機械と呼ばれるクラスの機械によって認識することができる．**図 7.3** の表は各形式的文法のクラスに対応する抽象機械の型を示している．

型	言語の型と認識を行なうオートマトン	生成則の形式と制限
0	狭義の文脈規定 (Postの体系): Turingの機械	$\alpha \to \beta$ $\alpha, \beta \in (N \cup T)^*$; $\alpha \neq \lambda$
1	広義の文脈規定: 非決定性 線型限定オートマトン	$\sigma \alpha \tau \to \sigma \beta \tau$ $\sigma, \tau \in (N \cup T)^*$; $\alpha, \beta \in (N \cup T)^* - \lambda$ および 長さ$(\alpha) \leqslant$ 長さ(β)
2	文脈自由： 非決定性プッシュ・ダウン 記憶付きオートマトン	$A \to \beta$ $\beta \in (N \cup T)^* - \lambda$; $A \in N$
3	正則または有限状態： [illegible]ートマトン	右線型 $A \to aB$, $A \to a$ 左線型 $A \to Ba$, $A \to a$ $a \in T$; $A, B \in N$

記号：X^* は空ストリング λ を含む集合Xから成るすべての有限ストリングを示す

図 7.3 基本的な形式的文法

7.5 BACKUS NAUR FORM—BACKUS 標準形—BNF

BNF とはプログラム言語の構文を記述するために一般に使用される文法を書く記法である．BNF においては非終端記号は角かっこ“〈”“〉”で囲まれた名前として記される．記号“→”は“::=”（“～で置換えられる”と読む）と書かれる．与えられた非終端記号を示す他の方法はたて棒“｜”（“または”と読む）によって区切ることである． BNF 記法を用いると前節の例1は次のように書かれる：

〈Σ〉::=〈A × B〉:　“記号Σは A に B を続けたもので置換えられる”と読む．

〈A〉::=a〈A〉| a:　“A は ‘a’ に A を続けたもの，または ‘a’ で置換えられる”と読む．

〈B〉::=〈B〉b | b

BNF の例として，図 7.4 に GO TO 文でのみできている FORTRAN 型の言語を定義する．GO TO 文は任意の長さの文ラベルと参照ラベルを持っている．もちろん，すべての参照ラベルは文ラベルの一覧表の中に現われることが

必要であるが，Backus Naur Form を用いて表現した言語にこの制限を課すことはできない．

〈文字〉::＝A | B | C | …… | Z —— （“文字はA，B，……，Zで置換えられる”と読む．）
〈記号〉::＝〈文字〉|〈文字〉〈記号〉
〈go to 文〉::＝GO TO〈記号〉
〈プログラム〉::＝〈記号〉〈go to 文〉|〈記号〉〈go to 文〉〈プログラム〉

図 7.4　FORTRAN 型言語のサブセットの BNF 定義

この言語で書かれたプログラムの例は；

```
   AB    GO  TO  XY;
   XY    GO  TO  WXYZA;
WXYZA    GO  TO  AB;
```

定義によるとラベルBやDが付いた文を持たない A GO TO B；C GO TO D；というようなプログラムを生成することが可能である．このようなプログラムを除外したいけれども，BNF の定義では形式的にこれを行なう方法がない．

〈プログラム〉の集合は正しいプログラムの集合よりもはるかに大きな集合である．BNF 記法は文脈自由（または句構造）文法と等価である．したがって，クラス記号は周囲の文脈に関係なしに展開される．すべての参照ラベルが文ラベルとして現われるようなプログラムのサブセットを区別することが望まれる．このような関係や機能を表現するために，より一層強力で，生成する文構造の要素間の相互参照を規定できるような形式的体系が必要である．相互参照は文脈規定型言語の特徴である．

7.6 正 準 体 系

本節では正準体系を言語定義のためのより強力な手段として説明する．筆者は正準体系が言語の理論的研究のための有力な道具であると考えている．現在行なわれている理論の研究の幾つかを例として説明し，幾つかの基本的な機能を用いて言語の定義を行なっている．

正準体系とは有限個のアルファベットでできているストリングの幾つかの集合に作用する形式的体系の1種である．正準体系は（Smullyan の**基本的な形式的体系**と等価であるが） Post の正準体系の1つの変形である．正準体系においては，生成の一般的な基本法則，すなわち，ストリング変換則の代りに公理（**カノン**）系および変数の代入と分離の論理的法則が用いられている．正準体系においては，相互に関連した述語の集合を定義する．各述語はストリングの集合である．

正準体系はプログラム言語の構文と翻訳を記述するために使用されてきた（More[3]; Donovan および Ledgard, 1967)．また，計算機用言語の汎用翻訳ルーチンのためのデータ・ベースとして役に立っている（Alsop, 1967; Altman[4])．後になって数学的な記述能力と形式的な性質に関する定理が証明され (Doyle[4], Mandl[4])，これらは翻訳ルーチンや言語の複雑さを研究するのに使用されている (Haggerty, 1969)．この研究の目標はプログラム言語に関する性質を記述しようとすることである．筆者はこの体系がプログラム言語やその翻訳ルーチンに関する諸性質の証明のために使用されるものと考えている．

Backus Naur Form (BNF) ではプログラム言語の構文を完全に記述することができなかった． BNF での“合法的なプログラム”は，形式的には正しくても，‘プログラム中の参照ラベルがすべて文ラベルに対応しない限りは翻訳プログラムに受理されない’といった，プログラム言語に存在する多くの制限を必らずしも満足していない．これらの特徴は“文脈規定の性質”と呼ばれる．一部の人達は，この特徴は言語の意味を述べており，構文よりむしろ意味論であると考えている．しかしながら，これらの特徴は言語の翻訳を定義するときに規定されなければならない．構文論と意味論との間には明確な区別がない．例えば，次の文

20 GO TO 20

は構文としては正当な文であるが，意味としては正しいであろうか．（この文は，計算機が誤りを発生する前にどれ位の間動作するかを決定するために有用

3 More, Yale 大学の講義ノート (1963) より．
4 未刊行 MIT 修士論文 (1970) より．

であるという人もあるであろう.)

プログラム言語の翻訳は，次の形式で順序付けられた対の集合を生成する正準体系によって規定される：

〈原始言語による文，目的言語によるその文の翻訳〉

このような規則の集合は計算機用言語の翻訳を規定する．もし目的言語が理解できれば，この定義は言語の意味を定義するともいい得る．

BNF と同様に，正準体系は言語のストリングを**生成**する．**認識**の過程は生成とは別の問題である．プログラム言語の正準体系による定義を用いて記号列を認識し翻訳結果を生成するプログラムが MIT で開発されインプリメントされている．

プログラム言語の文脈規定の部分を定義している正準体系は，言語の構文を記述しているのか，または意味を記述しているのかという議論はあまり意味のあるものではない．真の目的は言語を定義することによって言語とその翻訳に関する研究を行なおうとすることである．言語の翻訳を定義するために，翻訳が可能な正しい記号列のすべてを規定せねばならないというのでは困る．このような理由で構文上または意味上で不正な記号列を除外できるような強力で，BNF より能力の大きい体系が必要とされるようになったのである．しかし一般的な正準体系は能力が大きすぎ，その結果多くの非決定性の問題が発生する．したがって，正準体系の能力を正確に決定し，定義したい特徴を取り扱うのに適当な程度に，しかし処理できない問題を導き出さない程度に，その能力を制限しておきたい．

ここでは，まず，正準体系を通俗的に紹介する．正準体系は，一群の，‘ある**前提**はある種の**結論**を意味する’ということを述べる論理的な法則——カノン——から成っている．**述語**とは，目的言語におけるアルファベット記号列の集合で，意味が明確に定義された集合に付けられた名前である．1つのプログラム言語に対してこれらの集合は正準体系を利用するのに都合の良いように定義されている．

前提と結論を分離するために確言記号“⊢”を使用する．カノンの一般形は，

a ; b ; ....; c ⊢ z

と書かれ，“前提 a；b；………;c によって z が確言される”と読む．例えば，**数**の集合は次のようにして定義することができる：

⊢ 1 **数字**

⊢ 2 **数字**

⊢ 3 **数字**

x **数字** ⊢ x **数**

x **数字**；y **数** ⊢ yx **数**[5]

この体系は記号 1，2 および 3 からできる記号列の集合を**数**として定義する．変数 x および y に対して任意の終端記号を代入することができる．しかしその結果発生する前提が真でなければ，すなわち，その前提の宣言がそれ以前に結論として到達していなければ結論を得ることはできない．前提を持たない最初の 3 つのカノンは“公理”である．これらの結論は直接的である．

述語，変数，項，言，カノン

上の例では，**数字**と**数**は位数 1 の記号集合に名前を付ける述語である．位数 2 の述語は記号列の順序対の集合に名前を付ける．**項**とは変数と終端記号が結合した記号列であり，**言**(remark)とは述語記号が後に続く項である．$R_1, \ldots\ldots, R_{n-1} \vdash R_n$ がカノンとすると $R_1, \ldots\ldots, R_{n-1}$ は前提で R_n は結論である．各 R_i は言である．

以下では次の記号を使用する：

1．小文字は変数を表わす．
2．ゴジック体は述語記号として使用する．
3．記号 $\langle X_1 < X_2 < \ldots\ldots < X_n \rangle$ は n 項子を示す．位数 1 の項はかっこを付けずに単一の要素のみを記して示す．
4．同一の前提 $R_1, \ldots\ldots, R_n$ と幾つかの異なった結論 $\alpha_1 P, \ldots\ldots, \alpha_m P$ を持つ一連のカノンは次の省略形にて記される(**図 7.5** の例を参照)：

$$R_1; \ldots\ldots; R_n \vdash \alpha_{1+}\alpha_{2+}\alpha_{3+}\ldots{}_{+}\alpha_n P$$

---

5 カノンを書くときはゴチック体の代りに下線を用いることもある．
x <u>数字</u>；y <u>数</u> ⊢ xy <u>数</u>

形式的定義

定義12：　正準体系は次のような6項子である．

$$\mathscr{C}=(C,\ V,\ M,\ P,\ S,\ D)$$

ただし，

Cは**カノン**の有限集合である．

Vは$\mathscr{C}$で生成される（すなわち，証明可の）記号列を構成する終端記号のアルファベットである．

Mは**変数記号**（変数）の有限集合である．

Pはn項子の集合に名前を付けるために使用される**述語記号**（述語）の有限集合である．述語によって示されるn項子の中の要素の数は述語の**位数**である．

Sはカノンを書くのに使用される**区切り記号**の有限集合である．

D（⊆P）は**文述語**であり，これの合併は**正準体系によって定義される言語**である．

通常の区切り符号は ⊢，<，>，< および；である．

代入と分離

カノンの**適用**とはカノンに現われる変数に対して集合 V* の記号列を代入して得られる結果をいう．代入は不偏的でなければならない．ある1つの変数が置換えられるときには，つねに同じストリングで置換えられなければならない．推論の法則（**取り外し**とも呼ばれる）は，$R_1;\dots\dots\dots;R_{n-1}\vdash R_n$ がカノンの1つの適用であるとして，前提 $R_1;\dots\dots\dots;R_{n-1}$ がすべて体系に含まれているときのみ言 R_n が（正準体系において）導出できるということを述べている．“公理”の場合には前提がない．このとき R_n は $R_1,\dots\dots\dots,R_{n-1}$ から直ちに導出される．正準体系$\mathscr{C}$の言Rの**証明**または導出とは，各要素がそれに先行する1つ以上の言から直ちに導出されるような言 $R_1,\dots\dots\dots,R_n$, R の有限順序列をいう．Rが$\mathscr{C}$に含まれる（すなわち$\mathscr{C}$の中で導出されるか証明される）

ことと，Rに対して$\mathscr{C}$の中で証明が存在することとは同値である．

7.6.1 例： 構文の定義

本節では，プログラム言語の構文を記述するために正準体系を用いる例を示す．これらの例は Backus Naur Form の使用法を説明したとき用いたと同じ FORTRAN のサブセットを定義する．この同じサブセットの正準体系による定義を**図 7.5** に示す．

1 ⊢ $A_+B_+C\ldots\ldots\ldots_+Z$ 文字
2 *l* **文字** ⊢ *l* **記号**
3 *l* **文字**；*y* **記号** | *yl* **記号**
4 *y* **記号** ⊢ GO TO *y* **go to 文**
5 *i* **記号**；*x* **go to 文** ⊢ *ix* **プログラム**
6 *i* **記号**；*x* **go to 文**；*p* **プログラム** ⊢ *ixp* **プログラム**

図 7.5 構文の正準体系による定義

この定義を用いてプログラムを生成することができる．例えば記号列

A GO TO B

は，5番目のカノンの中に終端記号を代入して次のように生成することができる：

A **記号**；GO TO B **go to 文** ⊢ A GO TO B **プログラム**

公理(1)によりAは**文字**に属し，(2)によって**記号**に属す．したがって，カノン5の最初の前提は満足される．第2の前提はカノン4，2および1を用いることによって満足させることができる．したがって，次の手順は代入と分離を交互に用いて**プログラム** A GO TO B を導出する．(**図 7.6**)

1	A **文字**	C1, MP
2	A **記号**	1, C2, MP
3	B **文字**	
4	B **記号**	
5	GO TO B **go to 文**	4, C4, MP
6	A GO TO B**プログラム**	5, 2, C5, MP

(MP：分離；C*i*：図7.5のカノン； *k*=1, 2, …, 6：この図における行番号)

図 7.6 正準体系におけるストリングの導出

次に，同じ言語で‘すべての参照ラベルは文ラベルに現われること’という制限を持っている言語を定義する正準体系を構成しよう．したがって次のプログラムは，参照ラベルBとDは文ラベルAとCの中にないので，正しいプログラムではない．

A GO TO B
C GO TO D

この制限を有効にするために，ここで順序付きの3項子 $\langle$プログラム$_{<}$文照ラベルのリスト$_{<}$参照ラベルのリスト$\rangle$ の集合を生成しよう．この集合から参照ラベルとして文ラベルのリストに**含まれる**ものだけを持つプログラムを求めたい．このために，**図 7.5** の正準体系を次のように書き直す．

7　$\vdash A_{+}B_{+}C\ldots\ldots\ldots_{+}Z$ **文字**

8　l **文字** $\vdash l$ **記号**

9　l **文字**；y **記号** $\vdash yl$ **記号**

10　y **記号** $\vdash \langle$GO TO $y_{<}y\rangle$**参照ラベルを持つ go to 文**

カノン7～9は以前に用いられたものと同じである．しかし10は順序付きの集合を定義している点が異なっている．この集合の各要素は記号列 GO TO y および y から成っている．すなわち，各要素が参照ラベルを持つ GO TO 文と参照ラベルとから成っている．正しい GO TO 文に対してコードを生成するには，参照ラベルをも覚えておかなければならない．

11　s **記号**；$\langle x_{<}y\rangle$ **参照ラベルを持つ go to 文**

　　$\vdash \langle sx_{<}s,{}_{<}r,\rangle$ **文ラベルと参照ラベルを持つプログラム**

上述のカノンは順序付き3項子の集合を定義する．この集合の要素の第1成分は記号に GO TO 文が続くストリングから成り，第2成分は文ラベルで，第3成分は参照ラベルである．

カノン 11 の1つの適用は，

A**記号**；$\langle$GO TO MZ$_{<}$MZ$\rangle$ **参照ラベルを持つ go to 文** $\vdash$

<A GO TO MZ<A, <MZ,> 文ラベルと参照ラベルを持つプログラム

12 i **記号；**<x<l> **参照ラベルを持つ go to 文；**
<p<s<r> **文ラベルと参照ラベルを持つプログラム**
⊢<ixp<si, <rl,> **文ラベルと参照ラベルを持つプログラム**

カノン 12 は第1の成分が GO TO 文，第2の成分が文ラベルのリスト，そして第3の成分が参照ラベルのリスト（リストの各要素は“，”で区切られている）である順序付き3項子の集合を生成する．

13 <p<s<r>**文ラベルと参照ラベルを持つプログラム；**
<r<s>**包含** ⊢ P **プログラム**

上のカノンは，第1成分がプログラムであり，次の成分が文ラベルのリストで，第3の成分が参照ラベルのリストである3項子が与えられ，また，参照ラベルのリストが文ラベルのリストと包含関係にあるとき，プログラムは正しいプログラムであるということを述べている．次に，第1の成分がリストで，第2の成分が第1のリストのラベルを含むリストであるような順序対の集合としての**包含**関係を定義しよう．

カノン 14〜16 は**リスト**を定義する：

14 ⊢ λ **リスト**
15 i **記号** ⊢ i, **リスト**
16 x **リスト；** y **リスト** ⊢ xy **リスト**

カノン 17〜18 は述語：**包含**を定義する：

17 x **リスト；** y **リスト；** z **リスト** ⊢ <y<xyz> **包含**
18 <a<l> **包含；**<b<l>**包含** ⊢ <ab<l> **包含**

カノン 7〜18 は正しいプログラムの集合を定義する．

7.6.2 翻訳の定義

正準体系は言語の翻訳を定義するために使用できる．翻訳とは関数であり順序対の集合で定義することができる．この順序対の第1成分は正しいプログラムであり，第2成分は目的言語への翻訳結果である．例えば，PL/I の IBM 360 アセンブラ語への翻訳の定義は順序対の集合

〈正しい PL/I プログラム < 360 アセンブラ語のプログラム〉

をすべて規定する．

図 7.7 は FORTRAN の GO TO サブセットの IBM 360 アセンブラ語(BAL)への翻訳を定義している．例では簡単のために，参照ラベルが文ラベルのリスト中になければならないという制限を含まないし，実際の BAL が持つ名前の長さの制限をも持っていない．

⊢ $A_+B_+C_+$.........Z 文字
l 文字　⊢ *l* 記号
l 文字；*y* 記号 ⊢ *yl* 記号
y 記号 ⊢ <GO TO *y* < B *y*>翻訳付き go to 文
i 記号；<*x* < *y*>翻訳 go to 文　⊢ <*ix* < *iy*> 翻訳
i 記号；<*x* < *y*>翻訳付き go to 文；<*p* < *t*> 翻訳 ⊢ <*ixp* < *iyt*> 翻訳

図 7.7　FORTRAN のサブセットから BAL への翻訳の正準体系による定義

7.6.3　認識と翻訳のアルゴリズム

正準体系は，正しい記号列を生成する規則の集合によって言語を定義する．正準体系が，定義された規則の集合から生成される記号列を認識するための基礎となり得るかということは，興味ある問題であり，加えて，定義された集合の要素が順序対，3項子，などであれば，アルゴリズムは与えられた項に対応する欠除項を生成し得るかということも問題である．1つの直接的な認識アルゴリズムは，一致がとれるまですべての可能な記号列を単純に生成することである．本節ではより効率の良いアルゴリズムについて論じる．このアルゴリズムは Backus Naur Form 体系によって生成された記号列を認識する能力を持っている．このアルゴリズムは Cheatham と Sattley によって示されたアルゴリズムの拡張として Alsop によって開発された．

MIT のタイムシェアリング・システムに組込まれたこのアルゴリズムを用

いて，言語の正準体系による定義とその翻訳をタイプ入力し，次に言語の記号列をタイプ入力すると，プログラムは記号列を認識して，対応した翻訳結果を生成する．本章においては直構文コンパイラと回帰呼出しに用いられる技法を説明するためにこのアルゴリズムを述べる．

プログラムは2つの段階を持つ．予備段階では，使用する正準体系の構文を検査する．主段階では，入力記号列を走査し，正準体系の定義を満足するかどうかを決定し，対応した翻訳結果を生成する．アルゴリズムは基本的に**下降型**である．まず，入力記号列と正準体系の文述語（例えば，**プログラム**）との一致を探す．そして，回帰呼出しによって低位の述語（例えば，**整数**または**数字**）に到達する．

次に述べるのは，位数1の述語のみを含む正準体系に対する簡単なアルゴリズムの説明である．簡単化されたアルゴリズム（**図 7.8** 参照）は後に一般的な場合を含むように拡張される．

仮想的な指標が第1文字の左に設定されている任意の文字列と，記号列の集合を定義している正準体系を考える．ここで行ないたいことは文字列が集合の要素であるか否かを決定することである．

ステップ1　プログラムはカノンを順番に取り出し，処理しようとする記号列を直接定義しているかどうか調べるためにステップ2から6を実行する．

ステップ2　カノンの結論が入力記号列と一致するかどうか項目毎に調べる．結論の中の項目が終端記号であればステップ3が実行され，変数であればステップ4が実行される．カノンが終了すればステップ5が実行される．

ステップ3　結論中の項目が終端記号の場合はこのステップが実行される．この記号は入力記号列の指標の右の文字と比較される．もし等しければ，結論の次の項目を調べるために指標を1つ右に移してステップ2に戻る．等しくなければ走査は不成功であり，その記号列を生成する別のカノンを調べるためにステップ1に戻る．

ステップ4　結論中の項目が変数の場合はこのステップが実行される．プログラムは回帰呼出しを行ない，入力記号列の項によって変数の定義を決定しなければならない．すなわち，指標の右の文字から開始して，入力ストリングの文字数を決定しなければならない．そしてこの数が変数の定義として用いられ

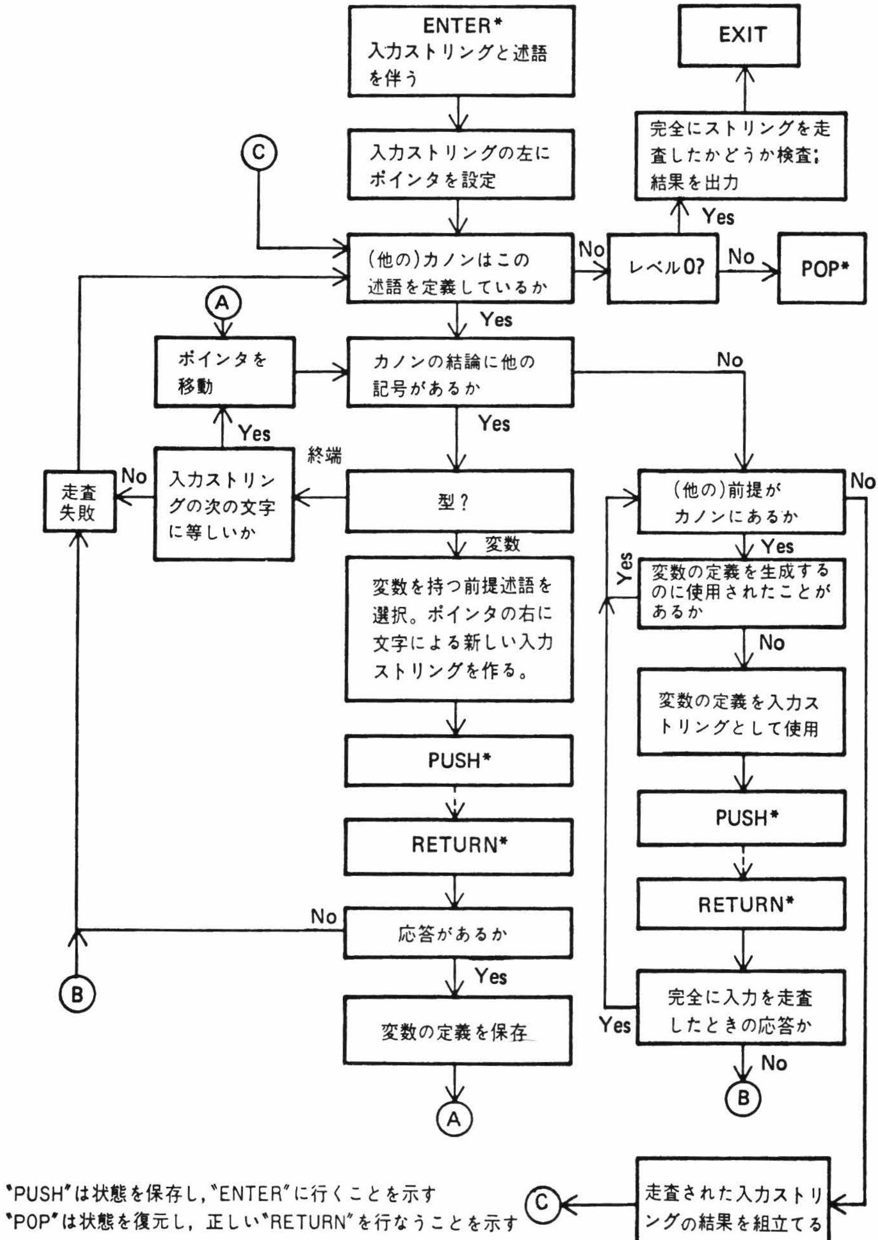


図7.8　簡単なアルゴリズムの流れ図

る．これを行なうために，プログラムは指標の右にあるすべての入力記号列を複写して新しい入力記号列を組立て，その変数を含むカノンの前提述語を探す．現在の状態をスタックに保存（“プッシュ”）してから，プログラムは，選ばれた前提の中の述語を定義しているカノンを調べて変数の定義を決定するために，ステップ1に戻る．プログラムが再び復帰してきたとき何も答えが返されなければ走査は不成功であり，記号列の他の定義を探すためにステップ1に戻る．答えが返されたときには，プログラムはこれと元の入力記号列を比較して変数の定義を決定し，指標を定義の次の位置に移す．アルゴリズムはステップ2に戻る．

ステップ5　結論の走査は完了し，結論に示される変数の定義が（入力記号列の形で）記録されている．アルゴリズムは次に前提を調べる．結論に含まれる変数の定義を決定するためにステップ4で使用された前提は，定義を生成するために使用されているので，既に確認されている．しかしながら，変数は前提の中に2回現われてもよいので変数の定義を形成する記号列は両者の集合の要素であることを保証しておく必要がある．アルゴリズムは変数の定義から入力記号列を形成し，変数を含む他の前提が同様に真であるか否か，すなわち，変数の定義を行なっている記号列は前提中の述語として名前が付けられた第2の集合の要素であるか否かを回帰呼出しを行なって決定する．復帰したときに何も答えがなければ，アルゴリズムは，前と同様他の定義を求めるためにステップ1に戻り，記号列の定義を探す．答えが返されたとき，プログラムは，記号列が完全に走査されたことを確認する．まだ判定のすんでいない前提が残っているときは，同様に処理を行なう．すべての前提が確認されれば，簡略化アルゴリズムは最終ステップに進む．

ステップ6　この段階での走査の結果が組立てられる．そして先に呼出しを行なっている1つ位数の大きい段階に対する応答を形成する．走査が不成功の場合には何も結果を作らない．成功した場合の結果は，結論の走査が完了した点に指標が置かれている入力記号列である．ここで，入力記号列がその要素であり得るような集合を直接定義するカノンが残っているときには，アルゴリズムはステップ1に戻る．各々のカノンは結果に影響を与え得るので，プログラムは，1つ位数の大きい段階において複数の応答が処理されるように，複数の

ステップ	回帰レベル	入力ストリング	使用したカノン	結　果	次の動作
1	0	↓31	4	—	数字をプッシュ
2	1	↓31	1	失　敗	次のカノンへ
3	1	↓31	2	失　敗	次のカノンへ
4	1	↓31	3	3↓1 数字	ポップ
5	0	3↓1	4	3↓1 整数	次のカノンへ
6	0	↓31	5	—	整数をプッシュ
7	1	↓31	1	失　敗	次のカノンへ
8	1	↓31	2	失　敗	次のカノンへ
9	1	↓31	3	3↓1 数字	ポップ
10	0	3↓1	5	—	整数をプッシュ
11	1	↓1	4	—	数字をプッシュ
12	2	↓1	1	↓1 数字	次のカノンへ
13	2	↓1	2	失　敗	次のカノンへ
14	2	↓1	3	失　敗	ポップ
15	1	1↓	4	1↓ 整数	次のカノンへ
16	1	↓1	5	—	数字をプッシュ
17	2	↓1	1	1↓ 整数	次のカノンへ
18	2	↓1	2	失　敗	次のカノンへ
19	2	↓1	3	失　敗	ポップ
20	1	1↓	5	—	整数をプッシュ
21	2		4	—	数字をプッシュ
22	3	↓	1	失　敗	次のカノンへ
23	3	↓	2	失　敗	次のカノンへ
24	3	↓	3	失　敗	ポップ
25	2	↓	4	失　敗	次のカノンへ
26	2	↓	5	—	数字をプッシュ
27	3	↓	1	失　敗	次のカノンへ
28	3	↓	2	失　敗	次のカノンへ
29	3	↓	3	失　敗	ポップ
30	2		5	失　敗	ポップ
31	1		5	1↓ 整数	ポップ
32	0		5	31↓整数	
				3↓1整数	終了

図 7.9　ストリング認識のステップ

結果を準備しなければならず，各々の可能性を検討しなければならない．次に示す例によってこの問題点を明らかにする．これ以上カノンがないときは，プ

ログラムはステップ7に進む.

ステップ7 プログラムは状態を"ポップ"し，1つ位数の大きい段階に復帰する．もし最高位の段階に達すれば入力記号列が完全に走査された結果が得られる．このとき入力記号列は始めに定義されている集合の要素である．もしそうでなければ，記号列に構文上の誤りが存在する．構文上で正しくない記号列がこのアルゴリズムによってすべて検出されるかどうかは明らかでない．一般にこの認識の問題は非可解である．

複数の解を含む処理とその問題点を次の簡単な例で説明しよう．

1 ⊢1 **数字**
2 ⊢2 **数字**
3 ⊢3 **数字**
4 d **数字** ⊢ d **整数**
5 d **数字**；i **整数** ⊢ di **整数**

このアルゴリズムを用いて記号列31が**整数**かどうか決定したい． この処理は**図7.9**に省略形で示されている．アルゴリズムの単純性と効率の低下をさけるために，カノンは右回帰性を許されない（例えば，次のようなカノンは許されない）．

i **整数**；d **数字** ⊢ id **整数**

図 7.10には位数1以上の述語を処理するアルゴリズムの拡張の概要を示す．このアルゴリズムに対しては1つ以上の項が未知のままで翻訳の結果が要求されている．

7.7 正準体系と形式的体系

Robert Mandl は正準体系と各種の形式的体系との関係に対する一般的な定理を証明した．

定理： すべての型の文法に対して，同一の言語を生成し，かつ対応するク

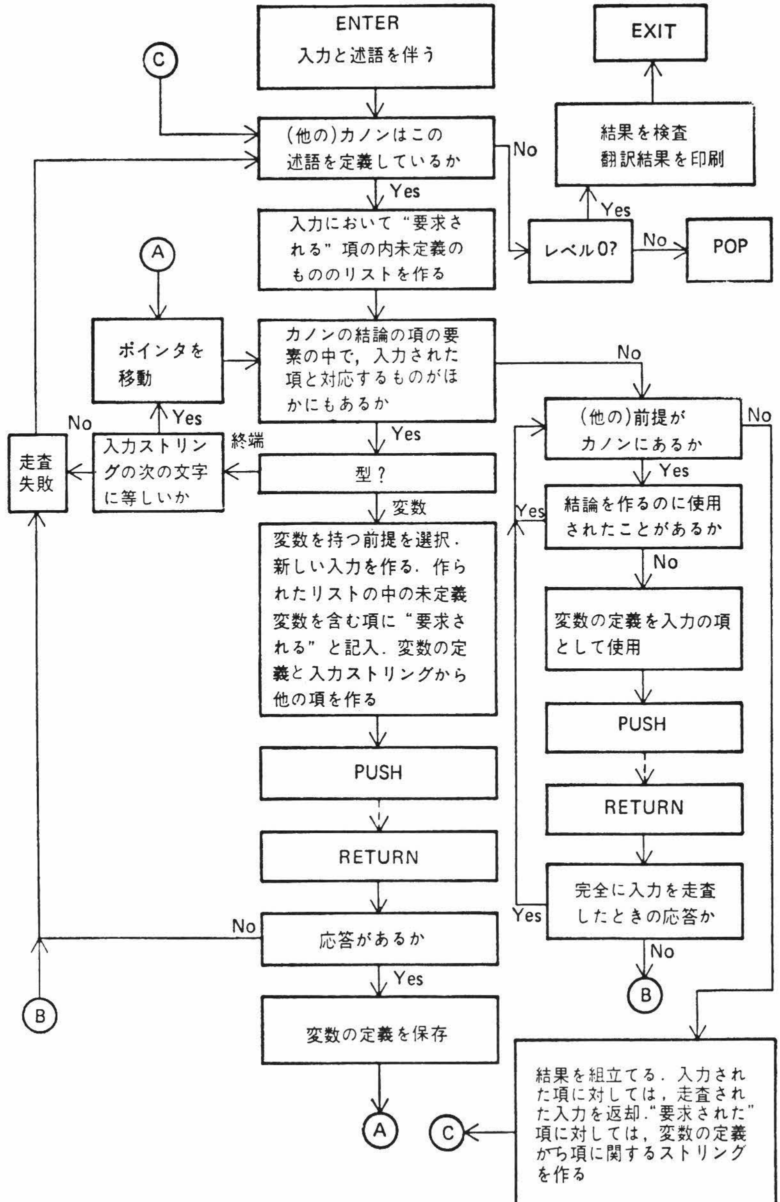


図7.10　不足項を認識し生成する一般的なアルゴリズムの流れ図

ラスに属する正準体系が存在する．さらに，そのクラスの正準体系を構成することができる．

ここでは正準体系の1つのクラスと線型文法との同値の証明の例を示す．最初，文法（N, T, P, Σ）の要素に対する正準体系（C, V, M, P, S, D）の要素の対応を確立し，次に，両体系が同一の記号列を生成することを示す．

また D＝{Σ} を選び Σ を**文記号（文述語）**と呼ぶ．

定義13： 各カノンが $\vdash aA$ または $xA\vdash axB$ であれば，正準体系は右線型単一前提正準体系（RLCS）であるという．ただし，
$x\in M$; $A, B\in P$; $a\in V$.

定理： すべての正則文法に対して，同一の言語を生成する右線型単一前提正準体系（RLCS）が存在し，逆にすべての RLCS に対して，同一の言語を生成する正則文法が存在する．同様の結果は左線型の体系（LLCS）についても成立する．いい換えると RLCS のクラスと正則文法のクラスの生成能力は等しい．

証明 1） $G=(N_G, T_G, P_G, \Sigma)$ を正則文法とし，次の RLCS を作る：

$$\mathscr{C}=(C, V, M, P, S, \Sigma)$$

ただし，

$$V=T_G$$
$$M=\{x\}$$
$$P=\{A\mid A\in N_G\}$$
$$S=\{;\,,\vdash\}$$
$$C=\begin{Bmatrix}\vdash aA & P_G \text{ の中で } \forall A\to a\\ xB\vdash axA & P_G \text{ の中で } \forall A\to aB\end{Bmatrix}$$

この文法と正準体系が同一の集合を生成することを証明するために，いずれか一方の形式的体系の導出が他の体系の導出によって遂一真似することができることを示す．証明は導出に使用されるステップ数に関して帰納法を用いて行なわれる．

帰納法の基礎　$\mathscr{C}$の1ステップの導出は $\vdash$aA の形でなければならない．そして，Aが$\mathscr{C}$の導出で確言されていれば，A → a はGの導出に適用可能である．

Gの1ステップの導出は A → a（これは“Aからの”導出を意味する）の形でなければならない．この生成が適用可能ならば $\vdash$aA は明らかであり，それは$\mathscr{C}$において対応する1ステップの導出である．

帰納のステップ　$K-1$ ステップまでの導出が，一方が他方によって模擬することができると仮定する．仮説によって $K-1$ ステップの正準体系の導出 ωB はGにおける証明 B⇒*ω に対応し，また逆も成立する．導出のK番目のステップにおいて，カノン $\vdash$aA（または生成 A→a）が使用されると，上述の“帰納法の基礎”で考察したと同様の条件となる．他の型のカノン（生成）の場合も同様である．模擬の仕方を2つの場合に分けて論じる．

a)　（上の文法によって模擬される正準体系の導出）導出のK番目のステップで xB$\vdash$axA の形式のカノンが使用されると仮定する．これが適用できるのは$\mathscr{C}$での証明　ωB　の存在を意味する．この導出は $K-1$ ステップを必要とするだけであるから，帰納の仮説によりG中で B⇒*ω である．

xB$\vdash$axA かつ $x=\omega$ の場合　$\mathscr{C}$中の証明 aωA を完結させる．G中の対応する規則 A → aB は A⇒*aω を与える．

したがって，$\mathscr{C}$中のすべての導出はGの中の導出，特にΣ導出を含めて，によって逐一模擬することができる．これで L($\mathscr{C}$)=L(G) が証明された．

b)　（正準体系によって模擬される文法の導出）Kステップの導出 A⇒*ω を考えよう．最初に適用される生成が A → aB であるとする．これが適用可であることは，Gの中で B⇒*ω' が $K-1$ ステップの長さであり，かつ，ω =aω' であるような記号列 $\omega' \in T^*_G$ の存在を意味する．帰納の仮説によってこのG中の $K-1$ ステップの導出は$\mathscr{C}$中の $K-1$ ステップの導出によって模擬できる．規則 A → aB に対応するカノンは

$$xB\vdash axA$$

である．

xに　ω' を代入すると$\mathscr{C}$の中のKステップの証明 ωA を得る．したがってG中で Σ⇒*ω であれば，$\mathscr{C}$の中に $\omega\Sigma$ が存在し，

$L(\mathscr{C})=L(G)$ である.

証明 2) $\mathscr{C}=(C, V, M, P, S, \Sigma)$ が RLCS であるとし,

$G=(N_G, T_G, P_G, \Sigma)$ を構成する. ただし,

$N_G=\{A \mid A\in P\}$

$T_G=V$

$$P_G=\begin{Bmatrix} A \rightarrow a & C\text{の中で } {}^{V}\vdash aA \\ A \rightarrow aB & C\text{の中で } {}^{V}xB\vdash axA \end{Bmatrix}$$

Σ は Σ に対応する.

この命題は $L(\mathscr{C})=L(G)$ を与える. 〔**証明終り**〕

なお, これらの体系は強等価（導出同志が等価）であることに注意してほしい.

定理：RLCS のクラスと正規文法のクラスは強等価である.

形式的体系と文法の研究の多くはそれらの生成能力と言語の構成ならびに複雑さに関してなされている. これらの性質は互いに関係がある. 本章の始めに複雑さの研究の動機について少し述べておいた. これら研究のすべてに共通することは, 生成的に使用でき, 生成的に解析できる形式にまで言語学を単純化しようということである. 言語学はあまりにも複雑多様であるので, 形式的体系に向わざるを得ない. しかし単に形式化することでこの問題が解決するわけではない.

多くの形式的体系, 例えば正準体系や 0 型文法などは固有の非決定性の問題を有している. 一般には有限時間内に与えられた記号列が与えられた文法の言語に属するか否かを示せるようなアルゴリズムは存在しない. （第 7.2 節の決定性の定義を思い出してほしい.）生成能力の研究により, 言語の構造的な特徴と文法の性質がいかに対応するかを知り, 与えられた条件に適する最も能力の小さい文法を選ぶことができる. 同時に, 制限を研究することによって言語の構造を学ぶことができる.

図 7.11 は文法のクラスの間の包含関係を示している. 図示された体系のクラスはそれより下に位置するすべてのクラスを含む. 分岐している枝は包合関

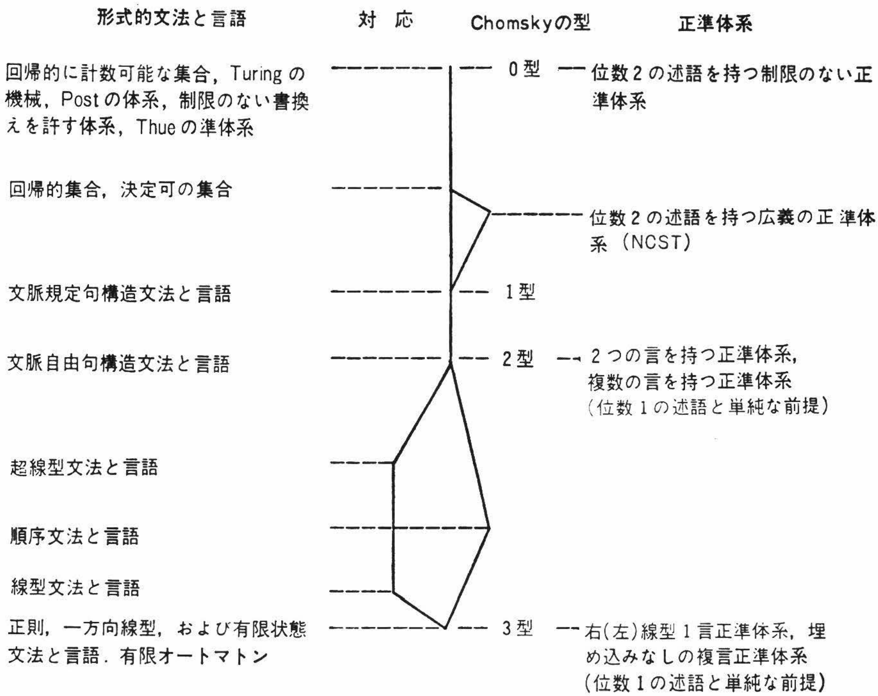


図 7.11　文法の階層構造

係が存在しないか現在までに解明されていないクラスを示している．

複雑さは直観的な概念である．これに対してはまだ満足すべき測度が提案されていない．式の“複雑さ”という言葉を用いて生成や認識における難しさのある種の測度を表現する．複雑さの**測度**は式や言語を比較するのに使用できる．測度として使えそうなものは証明の長さ，記号列の長さ，または導出の樹の“高さ”（枝の最も長い経路の長さ）である．正準体系の場合には証明で判定した言の個数を数えることである．これらの測度はすべて**構造**の面に関係したものである．構造を把握する別の方法は，形式的体系に対応する機械，すなわちオートマトンを考察することである．複雑さの測度は機械のある種の性質，例えば，所要記憶容量や処理のステップ数によって与えられる．

また，測度は意味や式の“深さ”――意味の概念――を表わすこともある．他の測度は式を認識したり，導出を構成することの難しさを表わす．後者の場合には，短い式や構造が単純な式は，もしそれがあいまいさを持ったり，他の多くの式に表面的にまたは構造的に類似しているとき，大きな複雑さの測度を持つ．

これらの測度は完全に一致しているわけではない．これらは言語の式，または言語そのものが異ると全く異なった順序付けを示す．言語翻訳システムの場合には異なった言語の等価な式の複雑さを考えることができる．例えば，与えられた翻訳のアルゴリズムが，ある測度に関して複雑さの順序付けを保存するかどうかを問うことができる．

式や文の構造は，文の意味に深く関係するから重要である（第2.7.1節の句構造の例を参照）．したがって，あいまいさは特別興味のあることである．（例えば，ある種の文脈自由言語はもともとあいまいである．すなわち，これらはあいまいでない文脈自由文法を持たないという定理がある．）2つの文法が共に与えられた文を生成するというとき，これらが文に同じ構造を付与したのかどうかをも知りたい．この質問は，複雑さが構造に関する概念であるので，複雑さに関係している．

7.8 まとめ

本章ではプログラム言語と言語プロセッサの定義，研究，および組込みにおける形式的体系の実用化を考察してきた．また**形式言語学**（文法，形式的体系および言語の研究）の用語を定義し，BNF は構文の記述と翻訳，およびプログラム言語の研究の手段として使用できることを示した．

この分野で現在行なわれている研究の例として，正準体系を示し，幾つかの形式的性質を考察し，その能力を図解した．文脈自由文法の1つである BNF によっては記述できない言語の性質が正準体系によって定義できることを示した．最後に，複雑さの概念を論じた．

演　習　問　題

1. 簡単な算術式 A+2*C の BNF による記述を考える.

(1)　〈式〉::=〈式〉+〈項〉|〈項〉

(2)　〈項〉::=〈因子〉*〈項〉|〈因子〉

(3)　〈因子〉::=〈記号〉|〈数〉

(4)　〈数〉::=0|1|2|3

(5)　〈記号〉::=〔〈文字〉〕$_1^8$

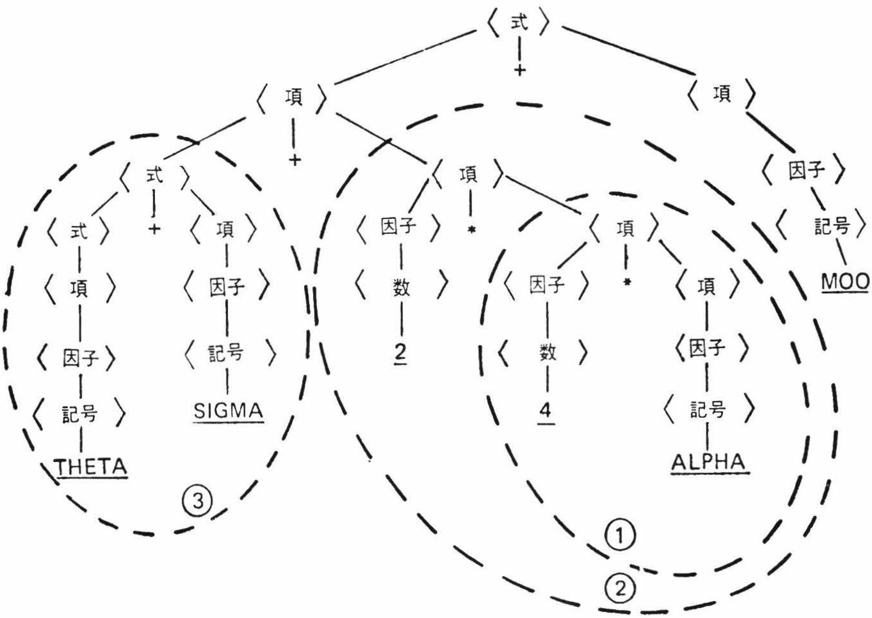
(6)　〈文字〉::=A|B|C|D

註：〔<文字>〕$_1^8$ は1文字から8文字までの文字列であることを示す. これは次の省略形である：

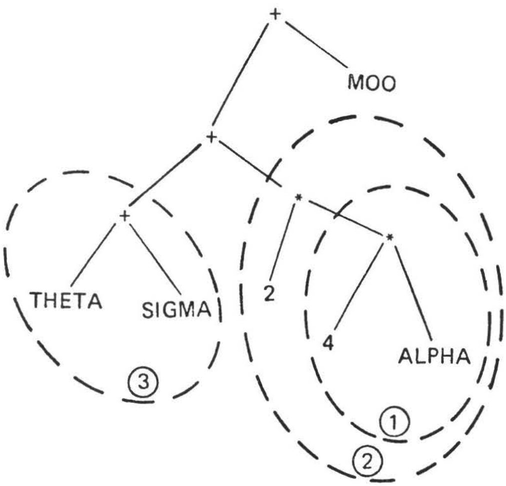
<文字>|<文字><文字>|<文字><文字><文字>|……

次式および“構文解析の樹”を考える.

THETA+SIGMA+2*4*ALPHA+MOO



上の樹の“簡略形”は次のように表わされる：



点線の円の各々は1つの節点から始まる部分樹を囲んでいる．これら3つの部分樹は BNF の左回帰性，右回帰性および優先順位の例を示している．優先順位とは部分樹における演算子の計算の順序である．

a. 部分樹①は，部分樹②との“＊”演算が行なわれる前に計算されなければならない．したがって，“2＊4＊ALPHA” において “4＊ALPHA” は最初に計算され，次に（2＊(4＊ALPHA)） が計算される．このように掛算は**右から左へ**の順で実行される．これは BNF の規則(2)の**右回帰性**に起因する．“::＝〈因子〉＊〈項〉” は，〈項〉は乗算を持ち得るが，“＊” の右側の入力であり，“＊” を行なう前に〈項〉の中の演算

が実行されなければならないことを示している．左側の〈因子〉は演算を含むことはできない．右回帰性によって計算は右から左への順で行なわれる．

b.　部分樹③は，上の BNF の規則(1)に起因する"＋"の**左回帰性**を示している．規則(1)で"::＝〈式〉＋〈項〉…"は，〈式〉は加算を持ち得るが，これが"＋"の左側の入力であり，"＋"を行なう前に〈式〉の中の演算が実行されなければならないことを示す．したがって，加算は左から右への順で進められる．左回帰性によって左から右へ演算子の計算が行なわれる．

c.　部分樹②は，演算"2＊(部分樹②の部分樹①)"が，この部分樹のぶら下っている"＋"が実行可となる前に計算されなければならないので，"＋"より"＊"の優先順位が高いことを示している．"＊"の優先順位が"＋"より高いということは BNF の規則(1)に示されている．〈項〉は乗算を持ち得るが，"::＝〈式〉＋〈項〉"によって〈項〉は"＋"の入力であり，"＋"より前に計算しなければならないことが指定される．一般に，演算子αへの入力〈人〉があり，〈人〉が演算子λを持ち得るとすると，λはαより優先順位が高い．

問

次の BNF で定義される言語を考える：

(1)　〈文〉::＝〈節〉△〈文〉|〈節〉

(2)　〈節〉::＝〈節〉□〈人〉|〈語〉$〈咳払い〉

(3)　〈人〉::＝〈語〉|〈言葉〉

(4)　〈語〉::＝〈語〉$〈文字〉|〈文字〉

(5)　〈言葉〉::＝〈文字〉△〈文字〉

(6)　〈咳払い〉::＝〈咳払い〉△〈文字〉|〈文字〉□〈文字〉

(7)　〈文字〉::＝A|B|C|D

a.　どの規則が回帰性を持つか．左回帰性か右回帰性かを示し，その回帰性を有する規則の番号とその部分を示せ．

回帰性と優先順位は関係がある．上の例では右の"＊"は左の"＊"より優先順位が高く，左の"＋"は右の"＋"より優先順位が高い．

b. 異なった演算子間の優先順位を述べている規則はどれか．その部分を示し，演算子間の優先順位を示せ．

c. b. を正しく答えた読者は，同じ種類の2つの演算子の間で反対の関係を見付けている．これは間違いでないし，あいまいさや矛盾を起こすものではない．その理由は何か．（ヒント：この関係はどのようなときに適用されるのか，いつも適用されるとは限らない．次の問を見よ．）

d. “A $ C □ D △ D” に対する “構文解析の樹” を示せ．

2. コンパイラやプログラム言語においてどのような面で形式的体系が使用されるか．例を4つあげよ．

3. a. 次の区別を述べよ．

1）文法

2）言語

3）計算機

b. 上のいずれが BNF 定義の性質を示すか．正準体系はどうか．

c. BNF と正準体系の関係を述べよ．

4. 生成文法（今考えている言語を記述できるものとする）で特定の記号列の正しさを決定するとき生じる困難は何か．

5. a. 言語を完全に記述する上で， BNF はなぜ不都合か．

b. 正準体系はこの欠点をいかに克服したか．

6. 次の Backus Naur Form の文法は記号列演算数と2項演算子を持つ “記号列表現” を生成する．これらの演算子によって示される変換は；

演算子	説　明	定　義
‖	結　合	ABC ‖ DE ‖ G ≜ ABCDEG
*	埋込み	AB * CD ≜ CDABCD
		X * AB ≜ ABXAB
-	出現を削除する	ABCDBABC - BC ≜ ADBA
		XX - X ≜ λ

BNF の定義は次のように与えられる：

〈記号列表現〉::= 〈部分記号列〉|〈記号列表現〉 ‖ 〈部分記号列〉

〈部分記号列〉::= 〈部分記号列〉 − 〈入れ子記号列〉

　　　　　　　　　　｜〈入れ子記号列〉

〈入れ子記号列〉 ::= 〈基本記号列〉

　　　　　　　　　　｜〈入れ子記号列〉 ＊ 〈基本記号列〉

〈基本記号列〉 ::= λ｜〈基本記号列〉〈文字〉

〈文字〉 ::=A|B|.........|Y|Z

λは null 記号列を示す

a. 正しい記号表現の例を2つ生成せよ（構文解析の樹を示せ）．すべての規則を少なくとも1度は使用せよ．

b. 最も優先位の高いものから順に演算子の優先順位を示せ（優先順位の定義については**図 7.1** を参照せよ）．

算術式 A+B＊C において，この文を構文解析する規則は＋を通過して＊を調べる必要がある．別のいい方をすると，規則は，Bが＋の右演算数ではなく＊の左演算数であることを識別しなければならない．

かっこのないときこの決定を行なうには，1つの演算数の両側の演算子を比べて，演算数がどちらのものであるかを調べなければならない．演算数は優先順位の最も高い演算子に属す．もし両方の演算子の順位が等しいときは左側のものに属する．2次元行列に演算子の優先順位を示すことができる．ただし左演算子とは演算数の左側にある演算子をいう．

c. 演算子の組に対して相対的な優先順位を示す2次元行列を作れ．

右演算子

左演算子	‖	＊	−
‖			
＊			
−			

7. 第8章（コンパイラ）の**図 8.16** に **PL/I** の小さなサブセットの **BNF** による定義が示されている．

a. 次の文の構文解析の樹を示せ．

COST＝RATE*(START-FINISH)＋2*RATE*(START-FINISH-100)

b. **図 8.1** のプログラムの構文解析の樹を示せ．

c. このサブセットの算術演算子（＋，－，*，/）の間に優先順位が定義されているか．式の演算はいかなる順序で行なわれるか．この順序はあいまいか．かっこの付いた式は正しいか．

d. BNF を拡張して通常の算術演算の優先順位を持つようにせよ．

8. 次は言語Lの正準体系による定義である：

(1) $\vdash$ $P_+Q_+R_+S_+$ **文字**

(2) i **文字** $\vdash$ j **記号**

(3) i **記号** $\vdash$ i **1次子**

(4) j **文字**；i **記号** $\vdash$ ji **記号**

(5) q **1次子** $\vdash$ q **項**

(6) t **項**；q **1次子** $\vdash$ $t*q$ **項**

(7) t **項** $\vdash$ t **式**

(8) t **項**；e **式** $\vdash$ $e+t$ **式**

(9) e **式** $\vdash$ (e) **1次子**

a. “＋” と “*” との間に優先順位の関係は定義されているか．されているとしたら，どのカノンがそれをどのように述べているか示せ．

b. Lはカノン(9)によってかっこに入れる表現を持っている． 幂乗の演算子“$”をLに追加せよ．“$”は“*”および“＋”より優先順位が高い．“$”は右から左へ演算しなければならない．したがって，

P ＋ Q * R $ S は P ＋ (Q * (R $ S)) の意味であり，
P ＋ Q $ R $ S は P ＋ (Q $ (R $ S)) の意味である．

9. 任意の BNF 体系は等価な正準体系で書くことができ，ある種の正準体系は等価な BNF 体系で書くことができる．問8の正準体系は BNF で書くこ

とができる．等価な BNF を書く方法の1つは；

1）各カノンに対して1つの BNF の規則を書く．

2）正準体系の述語（ゴチック体の名前）を BNF の非終端記号（“〈” “〉” に囲まれた名前）とする．

例

正準体系	**BNF**
⊢ $P_+Q_+R_+S$ **文字**	〈文字〉 ::= P|Q|R|S
j **文字** ⊢ j **記号**	〈記号〉 ::= 〈文字〉
j **文字**；j **記号** ⊢ ij **記号**	〈記号〉 ::= 〈記号〉〈文字〉

a.　上の例にならって問8の正準体系に等価な BNF を書け．

b.　正準体系は BNF（または文脈自由）言語より複雑な言語を表現することができる．これらの言語に等価な BNF 言語はない．次の言語に等価な BNF 言語は存在するか．存在するとすればそれを示せ．存在しないときはどのカノンと述語が BNF で定義できないかを示せ．

(1)　⊢ $A_+B_+C_+D$ **文字**

(2)　⊢ $0_+1_+2_+3$ **数字**

(3)　a **文字** ⊢ a **物**

(4)　a **文字** & b **物** ⊢ ab **物**

(5)　⊢ λ **リスト**

(6)　i **物** ⊢ i, **リスト**

(7)　a_+b **リスト** ⊢ ab **リスト**

(8)　a_+b_+c **リスト** ⊢ b **中** abc

(9)　a **中** m & b **中** m ⊢ ab **中** m

(10)　t **物** ⊢ GOTO t　**GO TO 文**

(11)　t **物** & s **物** ⊢ $s=t$ **ラベル割付け文**

(12)　s **GO TO 文** ⊢ s **文列**

(13)　s **ラベル割付け文** ⊢ s **文列**

10.　正準体系で定義される，言語や翻訳ルーチンに適用可能な複雑さの測度について述べよ．

11.　正準体系Cの定理 t の最も短い証明の中での言の個数を，正準体系Cの定

理 t の単純な測度 m(C, t) とする．すなわち，もし $r_1, r_2, \ldots\ldots\ldots r_k$ がCにおける t の最短の証明すると，m(C, t)=k である．

a.　t をCの定理とすれば m(C, t) は計算可能であることを証明せよ．

b.　新しい測定関数 m' を考える．

$$m'(C, n)=\text{maximum}\{m(C, t) \mid t \text{ はCにおいて証明可で，かつ，} |t|=n\}$$

いかなる条件下で m' は計算可能か．

c.　すべての正準体系はただ1つの前提カノンを有する正準体系に変換できることを証明せよ．

d.　この測度関数と正準体系Cによって定義される記号列 t を**認識**するためのステップ数との間の関係を求めよ．

12. a.　問10の測度関数に対して問 11.a および 11.b の定理を証明せよ．

b.　測度関数の上限および下限を求めよ．

索　　引

A

あいまいな文法 (ambiguous grammer), 258
あふれ (overflow), 232
アクセス (access)
　ポインタ使用——, 226（番地指定法，トランスファ・ベクトルの項も見よ）
アキュムレータ (AC) (accumulater), 110
ALGOL, 8, 216, 219, 230
アルゴリズム → プログラム，回帰性
ALLOCATE, 227, 236
アルファベット (alphabet), 252
ASCII, 3
アセンブラ (assembler), 4, 第 3 章
　データ・ベース, 69–72
　機能単位 (module), 84
　問題の設定, 65–68
　流れ図, 68–69, 80–81
　設計, 65–87
　処理の例, 76–79
　定義, 4, 17
アセンブラ語 (assembly language), 47–52
　プログラムの例, 47–52, 66, 88, 91–92, 108–109
　定義, 4（番地定数，リテラル，擬似命令，USING，DS，DC などの項も見よ）
アセンブル時 (assembly time), 7
"*"
　アセンブラ語の——（LC の現在の値）, 51, 74
　結合の——, 252
　ジョブ制御言語の——, 20
　PL/I の註釈文の——, 221
値 (value)，探索の——, 87
誤り（エラー）処理ルーチン (error handler)
　オペレーティング・システムの——, 13

B

バブル分類法 (bubble sort)→ 分類法
Backus Naur Form (BNF), 247, 261–262
　不適性, 263
　例, 282–284
倍精度 → 固定小数点，長形式
BAL（360 基本アセンブラ語）, 52（アセンブラ語の項も見よ）
BALR 命令, 49
番地 (location)，記憶場所，物理的な記憶装置の——, 3
　番地 (address), 3
　アセンブラによる——計算, 75–76
　ハードウェアによる——計算, 35
　ハッシュ法における——関数, 100
　インデックス・レジスタを用いた——修飾, 44
　実効 (effective) ——, 83
　命令をデータとして——修飾, 42
番地空間 (address space), 11
番地定数 (address constant), 29, 187
　アセンブラによる処理, 77–78
　ローダによる処理の例, 175–179,

190–194
複雑再配置 (complex relocatable) ——, 187
サブルーチン結合における——, 168
単純再配置 (simple relocatable) ——, 187
絶対 (absolute) ——, 187
番地指定法 (addressing)
ベース・レジスタによる——, 29
直接 (immediate) ——（SI 命令）, 36
直接番地指定型の——, 172
IBM 360/370 の——, 35（IBM 360/370 の項も見よ）
1 番地指定システム, 53–54
記号——, 47–50
PL/I の——, 219–220
例, 34
ベース・レジスタ (base register), 29
アセンブラによる選択, 83
割当て, 49
バッチ・ジョブ (batch job), 9
バッチ・モニタ・システム (batch monitor system)
発展, 9
——の例, 14
——の設計, 11–16, 19–22
バッチ・オペレーティング・システム (batch operating system), 9（バッチ・モニタ・システムの項も見よ）
バッチ制御言語 (batch control language), 13–16
BCD 2 進化 10 進コード (binary coded decimal), 3
BCR 命令, 52
バインダ (binder), 180, 213（ローダ方式の項も見よ）
ビット (bit), 3
再配置 (relocation) ——, 172–174
ブロック構造 (block structure)
使用法, 225–226
有効範囲, 225
BNF→ Backus Naur Form
ブール代数 (Boolean algebra), 247
BR, 14, 52
BSS→ ローディング方式
文 (sentence), 65, 258
文述語 (sentence predicate), 266
文脈自由文法 (context-free grammar), 259
分配分類法, 99（分類法の項も見よ）
文法 (grammar)
定義, 255
型, 259
分離 (Modus poneus; detachment), 263, 266
分類法 (sorting), 91–99
番地計算分類法 (address calculation sort), 97–98
バブル分類法 (bubble sort), 92
バケット分類法 (bucket sort), 92
分類法の比較, 98–99
分配法 (distributive), 99
比較法 (comparative), 99
互換分類法 (interchange sort), 92–93
基数交換分類法 (radix exchange sort), 95
Shell 分類法, 93–94
シフト分類法, 92
シンク分類法, 92
文書化の重要性 (documentation), 217
Burroughs 6500, 10
バイト (byte), 3, 28

C

CALL, 168

カノン (canons), 264–265
カノンの例, 267
CC→ 条件コード
超言語 (meta-language), 251
直構文コンパイラ (syntax-directed compiler), 248
直接演算数 (immediate operand), 36
直接結合ローダ → ローダ方式
直接的に導出されるストリング (immediately-derived string), 257
中間テープ（パスの間で使用される一時記憶）(collate tape), 157
中央処理装置 (CPU) (central processing unit), 3
CNOP, 132
COBOL, 8, 216, 230
コーディング技法 → プログラム技法
コマンド (command)→ 命令
COMMON, FORTRAN, 163, 174
コンパイラ (compiler)
　オペレーティング・システムとの関係, 8
　定義, 1, 8, 17
コンパイル時 (compile time), 7
コア (core), 5（記憶装置の項も見よ）
コア・ダンプ, 16
CSECT, 163
コア・イメージ作成ルーチン (image builder), 180（ローダ方式の項も見よ）

D

代入 (substitution)，正準体系の, 266
断片 (fragmentation), 10
データ (data), 3
データ構造 (data structure)
　使用法, 220–224
DC → 定数定義
DCL → DECLARE
DD 制御カード, 20
デバッグ (debugging), 16
デック, 64
　原始——, 162
　目的——, 162, 165
DECLARE, 221
デマンド・ページング (demand paging) → ページング
ディジタル計算機 (digital computer) → 中央処理装置
データ・セット (data set), 19
ディレクトリー (directory)，ファイル, 11
DO 文，PL/I の, 221
導出されたストリング (derived string), 254, 256, 257
DROP, 82
DS → 記憶域定義
DSECT, 109
ダンプ (dump), 16
ダイナミック・バインディング (dynamic binding), 181（ローダ方式の項も見よ）

E

EBCDIC, 3, 60
　変換の例, 60
エディタ (editor) → ローダ方式
END
　アセンブラで処理される——文, 50, 82–84
　ローダで使用される——カード, 177, 187
演算子優先順位 (operator precedence), 284–286
ENTRY, 168–170
EQU, 52, 55
　アセンブラで処理される——, 77
“=” → リテラル

=A（番地）, 78
=F（1 語長語）, 50
=H（半語）, 108
=V（外部番地）, 132
ESD（外部記号辞書） → ローダ方式
Euclid の幾何学, 247
EVENT, 234
EXEC, 14
EXTRN, 168–170

F

ファイル・システム (file system), 11
ファイル, 11
定義 (difinition), 1
FIXED BINARY, 236
フリッパ (flipper), 181（ローダ方式，オーバレイの項も見よ）
FORTRAN, 8, 216, 219, 220
FREE, 238（被制御記憶域の項も見よ）
浮動小数点数 (floating point number)
FORTRAN の——, 220
長形式 (long form) ——, 30
短形式 (short form) ——, 30
符号 (code) (BCD, EBCDIC), 3（命令コードの項も見よ）
複雑さの測度 (complexity measures), 280, 289

G

外部 (external)
アセンブラ語での参照, 168–170
ローダで処理される参照, 188
PL/I の属性, 237
外部記号辞書カード (ESD) (External Symbol Dictionary card), 177, 187, 196
記号, 173
合併 (merge), 95
GBLA (GloBaL Arithmetic), 160
GE 635 → HIS 635
GE 645 → HIS 645
言語 (language), 248, 251
アセンブラ——, 47–52
超 (meta) ——, 251
代数的 (algorithmic) —— → ALGOL, FORTRAN, PL/I
原始 (source), 5, 162, 163
ジョブ制御——, 14
形式的定義, 250–254
機械 (machine) ——, 39–47
——の記述 (specification), 248
高水準 (high level) ——, 第 6 章（高水準プログラム言語の項も見よ）
目的 (object) ——, 5, 162, 165（ローダの項も見よ）
原始プログラム (source program), 5, 162
限定文脈言語 (bounded-context language), 260
限定詞ポインタ (qualifier pointer), PL/I の, 226
GET（PL/I 読込み文）, 232
擬似命令 (pseudo-op), 49
擬似命令表 (POT) (Pseudo-Op Table), 70, 73
擬似乱数 (pseudo-random number), 103
語 (word), 28
GO TO, 237
GPL, 219

H

半語 (halfword), 28
判定 (probe), 102
汎用外部記号表 (GEST) (Global External Symbol Table), 194, 197–198

汎用レジスタ (general register), 25
ハードウェア (hardware), 25（命令，記憶装置，プロセッサ，2 次記憶装置の項も見よ）
ハッシュ法 (hash coding) → 探索
並列処理 (parallel processing) → 多重処理
変数記号 (variable symbol), 266
変数の有効範囲 (scope of variable)
　　DO ブロック, 236
　　条件 (condition), 232–234
　　開始ブロック (begin block), 225
　　記憶域 (storage), 224
　　手続き (procedure), 230
左回帰性 (left recursion), 284
左線型生成 (left-linear production), 260
非同期処理 (asynchronous processing), 4
比較分類法 (comparative sort), 99（分類法の項も見よ）
非決定プッシュ・ダウン・オートマトン (non-deterministic push-down automata), 261
引数表配列 (argument list array) (ALA), 135–137
HIS 635, 163, 170
HIS 645, 10
被制御記憶域 (controlled storage), 224
　　使用法, 224, 236
非終端記号, 252, 255
否定 NOT(¬), 222
保護 (protection)
　　セグメント――, 11
Honeywell → HIS
翻訳ルーチン (translator) → コンパイラ
保守，ファイルの → ファイル・システム
表の処理 (table processing), 87–103

I

IBM システム 360 と 370, 24
　　アセンブラ語 → アセンブラ語，アセンブラ
　　番地指定法 (addressing), 29
　　データ表現, 31
　　機械語, 39–47
　　機械語の例, 41–43
　　記憶装置, 28
　　ローダ → ローダ方式，直接結合
　　命令, 33, 36
　　モデル 40, 104
　　レジスタ (register), 29–30
　　例, 39–47
　　特殊機能, 36（番地指定法，レジスタの項も見よ）
IBM 360/370 アセンブラ語, 47
　　“拡張記号”, 107
　　例題プログラム, 47–51, 66, 88, 91–92, 108–109
　　マクロを用いた例, 156
　　問題, 58
IBM 370, 24
IBM 1130（ファイル・システム）, 163, 170
IBM 7094, 57, 163, 170, 186
ID（ローダにおける識別番号）, 188
IF···THEN, 221
IF···THEN···ELSE, 231
意味論 (semantics), 9, 250
インデックス・レジスタ (index register), 29
インデクシング (indexing) → プログラム技法
INITIAL, 236
入口と引数の宣言の例 (entry point and parameter declaration), 242
I/O チャネル, 26（I/O プロセッサの項

も見よ)
I/O プロセッサ, 11
　命令（コマンド）, 3
一般的な計算機の構造 (general machine structure), 24–26

J

自動記憶域 (AUTOMATIC storage), 224
時間 (time)
　コンパイル (compile)——, 7
　実行 (execute)——, 7
　ロード (load)——, 7
実効番地 (effective address), 83
実行時 (execution time), 7
辞引式順番 (lexicographical order), 89, 118
ジョブ (job), 9
JOB, 9
ジョブ制御言語 (JCL) (Job Control Language)
　例, 14, 21
　プロセッサ, 15
ジョブ・ステップ (job step), 14
情報検索 (information retrieval) → 表の処理
条件 (ON) (conditions)
　使用法, 232–233
条件コード (condition code), 56
乗積法 (power residue), 100
定数定義命令 (DC) (define constant), 48
純粋でない手続き (impure procedure), 4
純手続き (pure procedure), 4, 44
16 進数 (hexadecimal digit)
　定義, 32
　変換表, 32
述語 (predicate), 264
述語記号 (predicate symbol), 265
述語算法 (predicate calculus), 247

K

可変の表 (variable table), 76
開放番地法 (open addressing), 101（探索法，ハッシュの項も見よ）
回帰性 (recursion), 231
　例, 231, 271
　　正準体系での——, 271
　　マクロ・プロセッサでの——, 147（スタックの項も見よ）
解析的定義 (analytic specification), 251
階層構造 (hierarchy)
　正準体系，文法および機械の——, 280
　プログラム言語の——, 47
解釈ルーチン，インタプリタ (interpreter), 8
開始ブロック (BEGIN), 225, 236
開始記号 (starting symbol), 255
下降 (top-down), 271
仮説算法 (propositional calculus), 247
“_” 下線, 221
仮想記憶 (virtual memory), 11
形式的文法 (formal grammar), 255–258
　ストリングの導出, 257
　例, 254–256
形式的体系 (formal system), 8–9, 第 7 章
　定義, 8, 247
　階層, 258
　Post の体系, 8
　プログラム言語への応用, 247–250
　例, 256（Backus Naur Form，正準体系の項も見よ）
形式的定義 (formal specification), 250–

254
発展, 251–260（BNF，正準体系の項も見よ）
結合 → リンキング
結論 (conclusion)，正準体系の, 264
計算の可能性 (computability) → 決定可言語
決定可 (decidable), 251
キー → 見出し語
樹 (tree) → 構文の樹
基本的形式的体系 (elementary formal system), 263
機械 (machine)
番地 → 番地指定
解析 (analytic), 260
形式的――, 261
オートマトン (automata), 261
Turing――, 260（IBM 360/370, I/O プロセッサの項も見よ）
機械命令 (machine-op), 49
機械命令表 (MOT) (Machine Operation Table), 70, 73
計算機の構造 (machine structure), 24–26
新しい計算機を知るための一般的な手法, 26–28（IBM 360/370，I/O プロセッサの項も見よ）
記号 (mnemonic), 4
命令――, 47–49
機械語, 47
記号番地 (symbolic address), 47
記号コーディング (symbolic coding), 47
記号表 (ST) (Symbol Table)
アセンブラ――, 70, 74
コンパイラ―― → 記号表
帰納法 (induction), 277
記憶域 (storage)
コア → 記憶装置
――演算数 (operand), 34
管理 (management) → 記憶管理
オートマトン (automata), 261
記憶域定義命令 (DS) (define storage), 49
記憶域割付け (storage allocation)
ローダによる――, 202
記憶域割付け (storage assignment)
ローダによる―― → ローダ方式
記憶管理 (memory management)
断片, 10
記憶空間 (memory space), 11
記憶番地レジスタ (MAR) (memory address register), 26
記憶バッファ・レジスタ (MBR) (memory buffer register), 26
記憶制御装置 (memory controller), 26
記憶装置 (memory), 3, 5
基数交換分類法 → 分類法
構文 (storage), 9, 248, 250
構文解析 (parse, parsing), 248, 282–283
算術式 (arithmetic expression), 282
――の樹 (tree), 282–283
構文の樹, 253, 282, 283
高水準言語 (high-level language), 8
記述, 第 7 章（BNF，正準体系の項も見よ）
重要性, 217–219
特徴, 8, 219–220（PL/I の項も見よ）
固定小数点 (fixed point), 31
FORTRAN の――, 220
長形式 (long form)――, 31
短形式 (short form)――, 31
繰返し (iterate) → ルーピング
局所外部記号配列 (LESA) (Local External Symbol Array), 195, 198–199
局所定義 (LD) (local definition), 188
境界 (boundary)
DC または DS（1 語境界でなければならない）, 82–84

1 語長語, 82
2 語長語, 77
リテラル（2 語境界でなければならない）, 77
プログラム（2 語境界でなければならない）, 192–194
共有, セグメントの, 11（純手続き, セグメンテーションの項も見よ）

L

ラベル (label)
アセンブラの——, 50, 66–67, 76
——変数 (variable), 226–230
PL/I の——, 219, 220
LCLA(LoCaL Arithmetic), 160
ライブラリ (library), 16
リンケージ・エディット (linkage edit) → ローダ方式
リンキング (linking), 7
リスト処理 (list processing), 228
リテラル, 50（"=" の項も見よ）
リテラル表 (literal table), 51, 70
ローダ (loader), 5, 17
定義, 17
機能, 162
設計, 第 5 章
役割, 5–7
ロード・実行 (load-and-go) → ローダ方式
ロード・マップ, 194
ロード時 (load time), 7
ロード・モジュール (load module), 180
ローダ方式 (loader scheme), 163–183
バインダ (binder), 179
"コンパイル・実行"(compile-and-go), "アセンブル・実行"(assemble-and-go), 163–164
直接結合型ローダ (direct-linking loader), 174–179, 185–204, 210
ダイナミック・バインダ (dynamic binder), 179
ダイナミック・リンキング (dynamic linking), 181–183
ダイナミック・ローディング (dynamic loading), 180–181
エディタ (editor), 180
リンケージ・エディタ (linkage editor), 180
2 進記号サブルーチン (BSS) (Binary Symbolic Subroutine), 170–174
オーバレイ (overlay), 179
例, 190–194, 211–213
再配置型ローダ (relocating loader), 170
設計, 183–204
絶対型ローダ, 166–168, 183–185
呼出し時ロード (load-on-call), 181
ロケーション・カウンタ (LC) (location counter), 25
ロケーション・カウンタ (location counter), アセンブラの, 66
ルーピング (looping), 46–47
LTORG 擬似命令, 77

M

マクロ (macro), 第 4 章
言語, 第 4 章
擬似命令, 123, 128–129
位置決め引数, 127
条件付き擬似命令, 128–129
仮引数, 124
ラベル引数, 126
マクロ・ラベル, 128
マクロ定義の中のマクロ呼出し, 129
マクロ呼出し, 7, 124
命令, 124

命令の引数, 124
命令の名前, 123
見出し語付き引数, 127
問題の設定, 133
定義, 6, 7, 123
展開, 124（“マクロ・プロセッサ” の項も見よ）
MACRO, 123
マクロ名表 (MNT) (Macro Name Table), 135, 138
マクロ名表カウンタ (MNTC) (Macro Name Table Counter), 135, 148
マクロ定義レベル・カウンタ (MDLC) (Macro Definition Level Counter), 143–144
マクロ定義表 (MDT) (Macro Definition Table), 135, 137
マクロ定義表カウンタ (MDTC) (Macro Definition Table Counter), 135, 148
マクロ・プロセッサ (macro processor), 第 4 章
限定記号, 2 パスのアルゴリズム, 134–141
1 パス・アルゴリズム, 141
組込み, 132–155
アセンブラと共に, 154–155
マクロの中でのマクロ呼出し——, 145
問題の設定, 133
定義, 1, 17
命令 (instruction), 3
擬似命令, 49
位置決め引数, 127
書式
IBM 370, 36–39
単一番地指定
360/370, 36
3 番地指定法, 53
取り出し（記憶制御装置による）, 26
命令カウンタ (IC) (instruction counter), 25
命令解釈部 (instruction interpreter), 24
命令レジスタ (IR), 25, 53
MEND, 123
MFT, 10
マイクロ流れ図, 26
ADD 命令の例, 27
マイクロ・プログラミング（マイクロ流れ図のレベルでのプログラミング）(micro programming), 26–27
見出し語（キー）(key)
探索 (searching), 87
右回帰性 (right-recursion), 283
右線型生成則 (right-linear production), 260
右線型単一前提正準体系 (RLCS) (Right Linear Single Premise Canonic System), 277
モード (mode)
待ち——, 234
モジュール・ローダ, 180（ローダ方式の項も見よ）
文字表現 (character representation), 31
目的プログラム (object program), 5, 162
問題の設定, 65
モニタ (monitor), 14（オペレーティング・システムの項も見よ）
MQ（乗数／商レジスタ）, 110
MULTICS, 217

N

長さ，ストリングの (length of string), 259
長さの属性 (length attribute)，PL/I の, 223–224
Neuman, John von, 24

名前空間 (name space), 11（仮想記憶の項も見よ）
2 分探索法 (binary search) → 探索法
2 語長語, 28
　境界, 77
2 次記憶管理 (secondary storage management), 13（ファイル・システムの項も見よ）
2 進記号サブルーチン (BSS) (binary symbolic subroutine) → ローダ方式
2 進数変換表, 32

O

オートマトン (automata), 261（機械の項も見よ）
オフセット（番地指定の）(offset), 29
ON
　——ENDFILE, 232
　——OVERFLOW, 232（条件の項も見よ）
OS → オペレーティング・システム
オペレーティング・システム (operating system)（モニタ・システムと同義語）
　定義 (definition), 1
　発展, 9–11
　利用者の観点, 11
　設備 (facility), 16
　資源管理 (resource management), 10（資源管理，ファイル・システム，MFT，MULTICS，ページング，プロセッサ管理，再配置分割割付け，セグメンテーションの項も見よ）
オーバレイ構造 (overlay structure), 181（ローダ方式の項も見よ）
オーバレイ・スーパバイザ (overlay supervisor), 181

P

パックされない（10 進）数 (unpacked number), 31
パックされた 10 進数 (packed decimal number), 31
ページング (paging), 10
　デマンド (demand)——, 10
Peano の算法, 247
PL/I, 8, 216
　ブロック構造 (block structure), 225–226
　註釈 (comment) "/ * · · · * /", 221
　データの型やデータ構造の例, 220
　非同期動作 (asynchronous operation), 231
　柔軟な参照 (accessing flexibility), 220, 226–230
　条件 (condition), 232
　条件の例, 232
　回帰性 (recursion), 231
　拡張性とコンパイル時マクロ, 234
　記憶域のクラス, 224
　記憶割付けと名前の有効範囲, 224
　ラベル変数とラベル配列, 229（プログラム言語の項も見よ）
　文字列 (character string), 219
　見出し語 (keywords) → DCL, PROC, VARYING, IF, PROCEDURE など
　なぜ PL/I を選ぶのか, 216
　プログラムの例, 231, 236–238
　ポインタ (pointer), 226
　シグナル (signal), 233–234
　サブルーチン (subroutine), 230
　システム・プログラムへの使用, 217
　多重タスク (multitasking), 234
　手続き (procedure), 230
　割込み (interrupt), 231

POP, 271–275（回帰性の項も見よ）
Post, Emil, 255, 263
Post の正準体系, 255
PROCEDURE, 231
プロセッサ (processor), 3（CPU, I/O プロセッサ, IBM 360/370, 命令の項も見よ）
プログラム (program), 3
プログラム・カウンタ (PC) (program counter), 25
プログラム・ロード番号 (PLA) (Program Load Address), 193–194, 202–203
プログラム状態語 (PSW) (Program Status Word), 29
プログラム言語, 第 6 章
　機能／特徴 (feature), 第 6 章（PL/I の項も見よ）
　——記述 → BNF, 正準体系
　参照の柔軟性, 220, 226–230
　システム・プログラムにおける使用, 217–219
プログラム技法 (programming techniques)
　番地指定法 (addressing), 42–45（番地指定法の項も見よ）
　文書化 (documentation), 217
　連鎖 (chaining), 227
　インデクシング (indexing), 44
　回帰性（“繰返しは人間で, 回帰は神である”）, 231
　結合, リンキング (linking), 227
　機能モジュール (functional modularity), 230
　機能単位 (modularity), 230
　リスト処理 (list processing), 228
　リテラル (literal), 50
　ルーピング (looping), 46–47
　マクロ (macro), 第 4 章（アセンブラ語, 高水準言語, I/O プログラミング, 割込み処理, スタックの項も見よ）
プログラミング・システム (programming system)
　要素, 17
　発展, 4–9
プロジェクト MAC, 217
PSW → プログラム状態語
プッシュ(push), 271–275（回帰性の項も見よ）
PUT 編集（PL/I 出力文）, 232

R

乱数発生ルーチン (random number generator), 100
レコード (record), 65
RECURSIVE, 231
レジスタ (register)
　アキュムレータ (accumulator)——, 110
　ベース (base)——, 29
　インデックス (index)——, 29
　LC（ロケーション・カウンタ）, 25, 66
　命令 (instruction)——, 25, 53
　MAR（記憶番地）, 26
　MBR（記憶バッファ）, 26
　MQ, 110
　PSW, 29
　作業用 (working)——, 25
　360/370 の——, 29
連鎖, リスト処理の (chaining), 227–228
RETURN, 231, 236
RLD, 177, 187
論理積：AND(&), 222
論理和：OR, 222

S

再配置 (relocation), 7
　　——ビット, 172
　　動的—— → ローダ方式
　　——例, 42
　　ローダの機能 → ローダ方式
再配置分割割付け (relocatable partitioned allocation), 10
再配置・結合ディレクトリー・カード (RLD) (Relocation and Linkage Directory card), 177, 187
　　書式, 197 (ローダ方式の項も見よ)
再入可能コード (re-entrant code) → 純手続き
採点プログラム (grading program), 60, 239
作業用レジスタ (working register), 25
参照 (access) → アクセス
スケジューラ (scheduler), 11
セグメント (segment), 11, 163
セグメント定義 (SD) (Segment Definition), 188
セグメンテーション (segmentation)
　　定義, 11
正準体系 (canonic system), 262–275
　　複雑さの測度, 280
　　構文記述の例, 267
　　翻訳記述の例, 269
　　認識と翻訳のアルゴリズム, 270
　　他の例, 265, 287–288
　　他のシステムとの関係, 275
　　定義, 262, 266
生成則 (production), 255
生成的な定義 (generative specification), 251
正則言語 (regular language), 260
静的記憶域 (static storage), 224
設計手順 (design procedure), 65
線型文法 (linear grammar), 260
線型有界オートマトン (linear-bounded automata), 261
SETA, 156
SETB, 159
Shannon, C., 249
Shell, D. L., 93
初期プログラム・ロード番地 (IPLA) (Initial Program Load Address), 193–194
省略型文脈規定文法 (contracting context-sensitive grammar), 261
証明 (proof), 266
　　帰納法による (induction)——, 277
集合論 (set theory), 247
主記憶装置 (main storage) → 記憶装置
終端記号 (terminal symbol), 252, 255
シグナル (signal), 233–234
ソフトウェア (software) → データ，プログラム
測度関数 (measure function), 280, 289
装置 (device), 3
　　2 次記憶 (secondary storage)——, 13
スタック (stack), 147, 273
　　——計算機, 56
　　マクロの——, 147–153
　　——“ポインタ”，スタック計算機の, 57
　　スタック・ポインタ (SP), 147
START, 49–50
　　アセンブルの——, 66
静的 (STATIC), 224
スターリングの宣言 (Sterling declaration), 235
ストリング (string) (文字の有限のつながり), 250
　　文字列, 221
　　ビット列, 222

——結合, 221, 252
——長さ, 221
——操作, 221–223
サブルーチン (subroutine), 230
結合 (linkage), 168–170
開いた (open)——, 6（マクロの項も見よ）
閉じた (closed)——, 6（手続きの項も見よ）
推論の法則 (rule of inference), 266
SYSLIB, 20
SYSLIN, 20
SYSIN, 20
SYSPRINT, 20
システム編集 (system generation), 8, 121
システム・プログラム (system program), 1

T

多重プロセッサ・システム (multiple processor system), 10
スケジューリング → プロセッサ管理
多重プログラミング：固定タスク数 (MFT), 10
多重プログラミング：可変タスク数 (MVT), 10
多重タスク, 234（PL/I の項も見よ）
単純句構造文法 (simple phrase-structure grammar), 259
単純乱数法 (random entry with replacement), 101（探索，ハッシュの項も見よ）
重複を省く乱数法 (random entry without replacement), 101（探索，ハッシュの項も見よ）
単語解析プロセッサ (lexical processor), 242
例, 245
探索法 (searching)
逐次法 (linear)——, 88
ハッシュ法 (hash), 99–103
ハッシュ・コード (hash coding), 117–118
ハッシュ登録／探索, 99–103
開放番地法 (open addressing), 101
単純乱数法 (random entry with replacement), 101
重複を省く乱数法 (random entry without replacement), 101
2 分法 (binary), 89
システム・プログラムでの重要性, 105
対数法 (logarithmic techniques), 90
定義, 88
手続き (procedure), プログラム，ルーチン，サブルーチン, 3
PL/I の——, 230
タイム・シェアリング (time-sharing), 11
トークン (token), 242
トラフィック制御 (traffic controller), 11
トレーシング (tracing), 16
トランスファ・ベクトル (transfer vector), 171
ローダの—— → ローダ方式，BSS
トランスレータ (translator) → コンパイラ
Turing の機械, 260
TXT, 171, 188
書式, 197（直接結合ローダの項も見よ）

U

UNIVAC 1108, 163, 170
USING 擬似命令, 48–49
アセンブラによる処理, 66, 82–84

有限状態オートマトン (finite-state automaton), 261
有限状態文法 (finite state grammar), 261

V

VARYING, 222, 237

W

WAIT, 234
WATFOR, 163
割出し (fault), ソフトウェア割込み → シグナル

X

XDS 940, 10

Y

"→", 253
呼出し時ロード (LOCAL) (Load on CALl), 181（ローダ方式の項も見よ）
優先順位 (precedence), 284–286

Z

前提 (premise), 正準体系の, 264
ゾーン (zone), 31

<訳者略歴>

池田克夫（いけだ　かつお）

1937年　滋賀県に生まれる
1960年　京都大学工学部電子工学科卒業
1965年　京都大学大学院工学研究科博士課程学修退学
　　　　京都大学助手
1971年　京都大学助教授
　　　　文部省在外研究員（長期）としてユタ大学およびマサチューセッツ工科大学に留学
1978年　筑波大学教授
1984年　文部省在外研究員（短期）としてスイス連邦工科大学に留学
1988年　京都大学教授
　　　　京都大学大学院情報学研究科長（兼任）
1991年　大阪工業大学教授
現　在　大阪工業大学教授
　　　　京都大学工学博士，京都大学名誉教授，筑波大学名誉教授

システム・プログラム

ISBN4-87653-332-6

2002年 3月25日　初版印刷
2002年 9月25日　初版発行
2004年10月25日　2刷発行
2006年 9月25日　3刷発行
2008年10月10日　4刷発行

著　者　J. J. ドノバン
訳　者　池田克夫
発行者　村越勝弘
発行所　株式会社 科学技術出版
　　　　〒113-0033
　　　　東京都文京区本郷2丁目16-12
電　話　03-3813-6920　FAX:03-3818-7888
E-mail　mail@kagaku.co.jp
URL　http://www.kagaku.co.jp/
印　刷　三美印刷
製　本　松岳社

コンピュータアーキテクチャ

モリス・マノ　著／国枝博昭・伊藤和人　訳
A5判　550頁

ディジタル計算機の登場以来、処理の高速化と記憶容量の増大を目指してハードウェアの改良が続けられている。特に近年の集積回路設計・製造技術の進歩により集積回路のコストが低減したことで、大量の集積回路資源の利用が可能になり、マイクロプロセッサやメモリを含めたディジタル計算機のアーキテクチャは過去10年の間に大きく進化した。

しかしながらディジタルシステムの基礎理論は不変のものであり、どんなに複雑なディジタル計算機アーキテクチャであっても基礎理論の応用にすぎない。大規模かつ複雑化した最新の計算機アーキテクチャを理解し、あるいは設計するためにも、計算機システムの基礎の学習は依然として重要であり、しっかりとした基礎知識を身につけておく必要がある。

本書は、ディジタル計算機の基礎理論である論理回路および複雑なディジタルシステムの記述に不可欠なレジスタ転送言語を説明している。ディジタル計算機のハードウェアについては、中央処理装置(CPU)、メモリ、入出力装置などのディジタル計算機の構成要素、ならびにCPUのマイクロプログラム制御を解説している。ディジタル計算機のソフトウェアについては、アセンブリ言語およびアセンブラ、データ表現、算術演算方法を解説している。さらに、RISC、パイプライン、マルチプロセッサといった最近の計算機構成技術についても詳しく説明してる。本書によって、ディジタル計算機の理解と設計に必要な基礎理論を習得するとともに、最近の計算機構成技術に関する知識を得ることができる。

本書について

内容

第1章◎ディジタル論理回路
ディジタル計算機を理解するためには、まずディジタル回路の基礎知識が不可欠である。ディジタル回路の初歩として論理代数、組み合わせ回路、順序回路を説明する。また、論理関数の簡単化、組み合わせ回路の設計手法、順序回路の設計手法を解説する。

第2章◎ディジタル構成部品
ディジタル計算機の構成部品として一般的かつ重要なデコーダ、エンコーダ、マルチプレクサ、レジスタ、カウンタ、メモリについて、回路構造と典型的な利用方法を解説する。

第3章◎データの表現
ディジタル計算機では、様々なデータを取り扱う。ディジタルシステムが直接処理可能なデータは二値データであり、ディジタル計算機の内部では、複数個の二値データを組み合わせた符号によって数値データや文字データを表現する。整数、補数による負数表現、固定小数、浮動小数といった数値データの符号化方式と、文字データの符号化方式について述べる。また、数値データを用いる算術演算と符号化の関係について解説する。

第4章◎レジスタ転送言語とマイクロ操作
ディジタル計算機のような大規模なディジタルシステムでは、個々のディジタル信号の振る舞いを論理関数によって記述したとしても、その記述からシステム全体の動作を理解することは困難である。より抽象的な記述、すなわち、どのレジスタからどのレジスタへデータが転送されるか、転送の間にデータがどのように処理されるか、といった記述の方がディジタル計算機の動作の表現と理解に適している。データの転送および処理といったマイ

クロ操作によってディジタルシステムの動作を記述するための言語であるレジスタ転送言語を解説する。

第5章◎計算機の基本構成と設計
基本ディジタル計算機の構成と設計を説明する。基本ディジタル計算機は単純であるが、以降の章で説明する様々な機構を備えている。計算機の内部処理は、レジスタ転送言語を用いて記述されているので、レジスタ転送言語を丹念に読むことで、ディジタル計算機の内部動作を理解することができる。

第6章◎基本計算機のプログラミング
ディジタル計算機のプログラムについて解説する。ディジタル計算機が実行するプログラムは、計算機に処理を指令する二進符号の集まりである。計算機が直接理解可能な二進符号と、人間が理解可能な記号表現の関係を説明し、記号表現を用いたアセンブリ言語プログラムから二進符号を用いた二進プログラムへの翻訳処理であるアセンブラを解説する。また、入出力プログラミング、サブルーチン、割り込みについて説明する。

第7章◎マイクロプログラム制御
ディジタルシステムの動作制御では、構成部品に与える様々な制御信号が必要である。制御信号は、ハードウェアによって構成した制御回路によって生成する方式と、マイクロプロセッサが実行するソフトウェアによって生成する方式がある。後者のソフトウェアによる制御方式を、マイクロプログラム制御という。基本ディジタル計算機をマイクロプログラム制御するためのマイクロプログラム式制御装置の設計を解説し、典型的な命令セットのマイクロコードを書く方法を例示する。

第8章◎中央処理装置
中央処理装置（CPU）は、計算機の内部にあって二進プログラムの解釈およびマイクロ操作を実行する。典型的な構成の汎用レジスタと算術論理装置が共通バスで接続されたCPUの設計を解説する。CPUが実行する命令の書式とアドレスモードを説明する。また、簡略命令セット計算機（RISC）の概念を紹介し、その特徴と利点について解説する。

第9章◎パイプラン処理とベクトル処理
パイプライン処理とベクトル処理を用いることで、計算機の処理速度を向上することができる。パイプラインの概念と処理速度向上の仕組みをいくつかの例を用いて説明する。算術演算を高速化する算術パイプラインと命令実行を高速化する命令パイプラインの両方を解説する。RISCプロセッサで利用される効率良い命令パイプライン、遅延ロード、遅延分岐を説明する。また、ベクトル処理を利用するベクトルプロセッサとアレープロセッサを紹介している。

第10章◎算術計算
符号付き絶対値、符号付き2の補数固定小数、浮動小数二進数、および二進コード化十進数（BCD）のデータに対する加算、減算、乗算、除算のアルゴリズムを説明する。アルゴリズムは、レジスタ転送言語を用いたフローチャート形式によって表されている。また、ディジタルハードウェアによって算術演算を実現する方法を解説する。

第11章◎入出力構成
計算機には、コンソール、プリンタ、磁気ディスク、磁気テープといった様々な装置が接続されている。これらの装置とCPUやメモリの間でデータをやり取りするための入出力装置が計算機内にある。入出力装置が外部装置と相互通信する方法として、並列データおよび直列データの非同期転送方式について説明する。計算機内部で、入出力装置、CPU、メモリの間でデータをやり取りするためのプログラムI/O、割り込み起動転送、直列メモリアクセス、入出力プロセッサといった4つの転送モードについて解説する。

第12章◎メモリ構成
計算機内でデータを記憶するには、キャッシュメモリ、主記憶、磁気ディスクといった記憶装置を利用する。これらの記憶装置は、記憶容量やデータの読み書き速度が異なり、必要に応じて組み合わせて利用される。高速で小容量の記憶装置と低速で大容量の記憶装置から構成される記憶階層を解説する。キャッシュメモリとして用いられる連想メモリの回路構成とアクセス方式を説明している。また、実際の主記憶容量よりも大きなメモリアドレスを利用可能とする仮想記憶方式について解説する。

第13章◎マルチプロセッサ
複数個のプロセッサを同時に使用することで並列処理による処理高速化を図るマルチプロセッサについて説明する。複数個のプロセッサの接続構造、メモリアクセスのプロセッサ間調停を解説し、プロセッサ間データ通信と同期の必要性を説明する。また、個々のプロセッサが内部に有するキャッシュのデータ整合性問題を説明し、いくつかの解決方法を解説する。

◆コンピュータサイエンス・シリーズ　科学技術出版の本

分散オペレーティングシステム

プラディープ・K・シンハ 著／水野忠則 他 訳

A5判　1,030頁

高度情報化社会の発展を支えている重要な技術分野の一つとしてコンピュータサイエンスおよびコンピュータ工学があり、特に分散コンピューティングシステムの基礎となる分散オペレーティングシステムの技術がますます重要になってきている。

オペレーティングシステムは、単体で動作するスタンドアローン型のオペレーティングシステムからネットワーク機能を付加したネットワークオペレーティングシステム、さらには複数のCPUを使用したマルチプロセッサオペレーティングシステムなどへと進展してきている。
本書で紹介する分散オペレーティングシステムは、その中で最も進化したものであり、ネットワーク機能と純粋のオペレーティングシステム機能が完全に一体化したものである。ここで述べる分散オペレーティングシステムは、最新の技術的な概念を含んでいる。本書は、このような分散オペレーティングシステムのすべてを網羅し、かつ体系的にとらえ、分散オペレーティングシステムが何であるかを明確に解説している。

これまで、この分野の多くの本は特定の一研究分野に絞って解説し、分散オペレーティングシステムの全体を理解することは複雑で、時間を要するものであったが、本書は、分散オペレーティングシステムに横たわる基本的な概念と設計原理を分かり易く説明している。また、特定の分散オペレーティングシステムやハードウェアに限定していないために、各種の分散オペレーティングシステムに適用可能な、基本的な概念と設計原理が、一般的に記述されている。また、事例研究として、実際にこれまで研究開発されたきた分散オペレーティングシステムについても詳しく述べている。

特に、本書の特徴の一つとして参考文献が充実しており、インターネットのホームページ参照ポインタによる参考文献が記載されている。これにより、現時点での分散オペレーティングシステムに関する技術が容易に入手できるようになっており、さらに、インターネットを利用することによって、本書以降の最新の状況をも入手することができる。さらに、各章の最後には豊富な演習問題が載っており、それを解くことにより、レベルの高い理解度が得られように構成されている。

したがって本書を勉強することにより、ネットワークを介した高度なコンピュータシステムを実現する各種の技術を取得でき、実際の実装、そして新たな技術への展開をはかることが可能であり、この分野の最新技術を学ぶ技術者や研究者にとって有益な著書である。

本　書　に　つ　い　て

構成

本書は12章から構成されている。

第1章は、分散オペレーティングシステムに関する基本的な技術についての導入章である。分散コンピューティングシステム、分散オペレーティングシステム、および分散オペレーティングシステムの基本技術についての特徴を解説し、歴史的な背景とともに分散コンピューティングシステムのモデル、さらに本書の中心的テーマである分散オペレーティングシステムの設計の課題について分かり易く記述している。

また、本書において事例研究として紹介されている分散コンピューティング環境（Distributed Computing Environment: DCE）についても簡単な解説を行っている。

第2章では、コンピュータネットワークの簡単な紹介を行うとともに、最新のネットワーク技術を述べる。
分散システムは基本的にハードウェアやソフトウェアの資源をもつ複数のノードから構成されるコンピュータネットワークである。このような分散システムの性能や信頼性は、コンピュータネットワークの性能や信頼性に依存ため、分散オペレーティングシステムを学習するためには少なくともコンピュータネットワークの基本知識が必要となる。本章では分散オペレーティングシステムの設計の基礎として必要となるネットワークを構成するための概念と設計について解説している。具体的には、ネットワークのタイプ（LANとWAN）の特徴、LAN技術としてのCDMA/CDプロトコル、トークンリングプロトコル、さらにLAN技術の応用事例、WAN技術としての交換技術、中継技術、通信プロトコル、イーサネットプロトコル事例について述べており、この章を理解することで基本的なコンピュータネットワークについての基本技術を十分理解することができ、コンピュータネットワークについての入門書としてもこの章を活用できる。

第3章では、分散オペレーティングシステムの重要な機能の一つであるプロセス間通信機構の設計に必要となるメッセージパッシング機構について説明している。

第4章では、分散アプリケーション共通に使用できる汎用的なメッセージパッシングモデルに基づくプロセス間通信（IPC）プロトコルとしての遠隔手続き呼び出し（RPC）の機能について説明している。

第5章では、メッセージパッシングと対称的な情報の共有方法の一つであるプロセスに共有アドレス空間を提供する分散共有メモリ（DSM）の機能について述べている。
この分散共有メモリ方式では、プロセスはこの共有アドレス空間を通常のローカルメモリを使うのと同じ方法で使うことができる。

第6章では、分散システムにおいて並行に動作する複数のプロセス間で生じるシステム資源の共有に関わる同期機構について説明している。

第7章では、ネットワークによって相互接続された分散システムは多くの資源を含んでおり、これらのシステム資源に関する資源管理の機構について資源の多重性およびシステムの透過性の観点から述べる。

第8章では、前章で述べた資源管理を受けて、分散オペレーティングシステムにおけるプロセス管理の機構について述べる。

第9章では、分散環境におけるファイルシステムである分散ファイルシステムの設計と実装について説明する。

第10章では、分散システムにおける位置透過性の目標を達成する上で非常に重要な役割を果たす名前付けの機能の設計および実装についての各種の取り組みを述べる。オブジェクトの複製を扱う場合にはさらに進んだ抽象化レベルを提供し、さらにオブジェクト共有をも実現しており、ここでは透過的移動、オブジェクトの複製の実現と共有に関する設計と実装について説明している。

第11章では、分散システムにおけるセキュリティの機構について述べている。
コンピュータシステムのセキュリティの目標は、そのシステムのセキュリティ方針によって決まり、それらの目標を達成するための方法はセキュリティ機構と呼ばれる。コンピュータセキュリティの共通目標は、安全性、プライバシ、信憑性、完全性であり、この点についての機構を説明する。また、分散システムにおけるセキュリティを提供するために通常用いられている機構について説明している。

第12章では、 具体的な分散オペレーティングシステムについて、これまでの章で述べた概念に関連する4つの実際の分散オペレーティングシステムの事例研究について述べている。ここで取り上げている事例を理解することによって、これまでの章で紹介した様々な概念について読者はより一層の理解を深めることができる。特に、各システムの設計の目的、システムアーキテクチャ、そして重要で価値のある特徴についてのみ焦点を当てて説明している。

また、参考文献には他の幾つかのシステムに関する資料が列挙されており、これらのシステムも分散オペレーティングシステムに関する研究活動を行っている人にとって興味深いものであろう。

理工系のためのC言語プログラミング入門

内田智史 著　A5判　430頁

一般にプログラミング言語を習得することは大変なことである。自由にプログラムを組めるようになるには、ハードウェアやソフトウェアの知識だけでなく、かなりのプログラミング経験を積まなければならない。その中でも特に、C言語は難しいと言われている。「学問に王道なし」と言われているように、C言語を習得して、自由にプログラムが組めるようになるには、大変な量の学習が必要になる。本書は、C言語の習得が大変難しいことを十分に認識した上で、まったくの初心者を対象にして、その読者がプログラムを確実に組むことが出来るようになるまでを解説したC言語のベストセラーの著者によるC言語によるプログラミングの演習用テキストである。また、本書で取り上げている例題は、極力、理工系の科学技術計算を中心としたものにしている。本書のレベルは、大学低学年の学生が読んで、十分自習できる内容に設定している。各章では、最初に、その章で解説する文法事項の全体像を説明した後、問題を解く形式でC言語のプログラミングについて解説している。そこでは、(1) 問題、(2) 分析・設計、(3) プログラミングの順で解説を加えている。さらに、必要に応じて、(4) 考察、を設け、より高度な話題を提供するように構成されている。分析・設計の項では、プログラム作成過程をできるだけ明確に読者に伝えるように工夫されている。プログラムを作成する場合、どんなに優秀な人間でも、すんなりとプログラムを完成させることはまれで、必ず、失敗やミスが付き物である。ところが、最終的に完成したプログラムは、当然、デバッグ作業により修正されてしまうから、そのような失敗の痕跡が消えてしまっている。本書では、そのような失敗の過程もできるだけ、読者に伝えるように配慮されている。

前半の章では、可能な限り説明を詳細にして読者の理解を高め、章が進むにつれて説明をシンプルにして読者の煩わしさを排除している。また、本書では、説明のためだけの無意味な例題を載せることは極力避け、意味のある例題を載せるように工夫されている。また、できるだけ理数系の学生の興味を引くような例題を選択している。各章の最後には、演習問題が配置されている。この演習問題を出来るだけ多く解くことがC言語習得の秘訣といえるであろう。演習問題も、あまり現実ばなれした意味のないものは載せずに、読者がこれからプログラムを作成していく上で役に立つような項目を取り入れるように配慮されている。また、なるべく読者がC言語のもついやらしい落し穴にはまり込まないように注意した。C言語の文法は、JIS規格に沿うように配慮している。用語もJIS規格に沿うように記述されているが、いくつかの用語については、JIS規格の用語ではなく、一般的に知られている用語を用いている。また、本書では、特定のプラットフォームやコンパイラに依存しないように配慮されているが、具体的な処理系を想定しなければならない部分では、GNU Cコンパイラを仮定している。

本　書　に　つ　い　て

構成

第1章「やさしい入門」
初心者を対象にプログラムやプログラミングの概念を紹介している。

第2章「変数と計算」
基本的なデータ型とそれを用いた計算プログラムの作り方について解説している。プログラムを組むときに意外に気づかないオーバーフローの問題や計算誤差についても取り上げている。また、組込み関数の使い方について

も言及している。

第3章「条件判断」
if文などの条件文の解説とそれを用いたプログラミングについて解説している。組込み関数のエラー対策などについても言及している。

第4章「ループ」
繰り返し処理について説明している。繰り返し処理は、タイミングの概念を含むので、初心者は最初にここでつまずきやすい。そのため、非常にやさしいところから解説している。また、科学技術計算のシミュレーションなどによく用いられる乱数についても取り上げている。また、プログラムをデバッグする際に必要となるシンボリック・デバッガについても解説している。

第5章「配列」
主記憶装置内で大量のデータを扱う配列について解説する。これによりベクトルや行列を扱えるようになる。配列を使えるようになり、より本格的な処理が可能になったので、演習問題のレベルも第5章からは少し高くなっている。

第6章「関数」
C 言語のプログラム単位である関数について学習する。分割コンパイルなどについても言及している。これにより、大規模なプログラムを作成することができるようになる。そのために、段階的詳細化などのプログラムの作成技術についても解説している。また、再帰関数を扱うことができるようになったので、クイックソートなどのテクニックも紹介している。

第7章「ポインタと構造体」
ポインタとそれに関連するプログラミングテクニックを学習する。配列とポインタとの関係やメモリの動的割り当てなどについても説明している。構造体を学習した後には、データ構造の基本であるリスト処理について言及している。リスト処理では、その考え方やプログラムの作り方についても説明している。

第8章「ファイル」
補助記憶装置上に確保された大量のデータを処理する方法について学習する。テキストファイルだけでなく、効率よくデータを処理できるバイナリファイルについても言及している。

第9章「練習問題のヒント」
各章の「練習問題」のヒントが詳細に記載されている。本文中ではピンとこなかったプログラミングテクニックが、このヒントを読むことによって明らかになる。

本書が、他のC言語の書籍と異なる点は、読者にそれなりの努力を要求しているという点である。前述したように、プログラミング言語の習得、その中でも特にC言語の習得は非常に大変な学習を必要とする。そのために、可能な限り多くの演習問題を配置して、読者が自ら学習できるように配慮されている。読者の確実な努力を要求するため、演習問題の解答は記載していないが、その代わりに詳細なヒントをすべての演習問題に記載している。このヒントの中には、プログラミングテクニックがさまざまな形で紹介されているので、読者のプログラミングテクニックの向上に必ず役に立つであろう。本書は、一通りC言語のプログラミングで必要になるであろうと思われる項目は網羅している。本書は、演習書に近い形態を取っているため、プログラミングの考え方やテクニックは詳細に説明している。本書はあくまでも初心者に対する入門書である。本書の読者層として期待している読者は、本当にプログラムを組むことが出来るようになりたいと願っている読者である。本格的な例題と演習問題を多く含めているのはそのためである。C言語は、相当努力しないと、自分のものにすることができないが、本書は、そのような読者にとっては必読書と言える。

第5章は常微分方程式の数値解法の基礎を初期値問題と境界値問題に分けて解説した。また常微分方程式の解法の偏微分方程式への応用である線の方法も取り上げた。

第6章は連立1次方程式の解法、第7章はその応用である固有値問題について解説した。

付録では、張力スプライン、ガウスの積分法および高速フーリエ変換について概要を述べている。

構成

本書の特徴は、数値計算の理論（理論編）と数値計算の実際（実際編）の2部構成としたところにある。

理論編では、数値計算法のしくみを、できる限り枝葉をとりその骨格がわかるようにしながらも、記述は平易となるように解説されている。すなわちコンパクトでありながらもわかりやすい本となっている。ここでは具体的な数値例についても最小限にとどめ、「泥臭く」ならないように配慮されている。

実際編では、理論編に述べた方法を実際のプログラムにするとどのようになるかを具体的に示している。そのために理論編で示した計算法を、実際のプログラムに組みやすいようなアルゴリズムの形でもう一度示している。また実際の数値計算の雰囲気をつかんでいただくため、数値例も多く示して、例えば収束計算では正解に近づく様子などがわかるようにした。

章だては1章を除き理論編と対応するようになっている。すなわち、理論編を読んでいてプログラムがどうなっているかとか、具体的な数値例が知りたければ実際編の同じ番号の章、節を参照すればよいようになっている。ただし、本編でプログラムを示した目的は計算法の考え方を知るためであり、それらをライブラリとして使うためのものではない（よいプログラムは広く流通している）。したがって言語としては数値計算用に開発されプログラムの構造がわかりやすいFORTRANと、最近情報系の学科で主力言語として使われているCが用いられている。なおCについても、C特有の使い方は避け、冗長であっても構造がわかりやすくなるようにしてある。

Javaで学ぶ数値計算

吉川　浩 著　　A5判　260頁

本書は数値計算プログラミングの基礎について、言語としてJava を用いて解説している。対象は大学1年生程度以上を想定しているが、コンピュータに関心がある中学生、高校生でもある程度の理解は可能であろう。Javaプログラミングの入門書はすでに多く書籍が出版されている。しかし、それらの多くはCの知識を前提としていたり、WEBページ装飾のためのアプレットやオブジェクト指向に重点をおいて基礎的な事項の説明が充分でなかったりする。また、基礎から丁寧に解説している書籍では多くの内容を含めるために分厚いか、分冊となっている。それに対して、本書はプログラミングの経験の無い読者を対象に、初歩的な事項からプログラミングの解説をおこなっている。また、アプレット（あるいはスタンドアローンアプリケーション）によるグラフィカルな表示は実用的な数値計算に役立つ範囲にとどめ、全体を適度なページ数に抑えたので、大学等での半期での講義にも有用である。想定する読者のコンピュータに対する予備知識としては、ワードプロセッサの操作ができる程度とした。

プログラミングの説明に入る前に、Javaのソースファイルの入力からコンパイル、実行までの手順を一通り説明している。また、フリーのJava開発キットJDK1.2（Java 2）Windows版のインストールと設定、使用方法についても解説している。

また、後半部分は数値計算入門になっていて、実用的な例題を多数示しながら、実際にプログラムを組むときの考え方を交えて説明をおこなっている。講義での使用を考え、演習問題などを適宜挿入する。

なお、内容については同一著者による姉妹書「JavaScriptで学ぶ数値計算」と類似の部分が多く、効果的に両書を学ぶことができるように工夫されている。

本　書　に　つ　い　て

内容

第1章では、プログラミング言語である Java の特徴や、C/C++、Fortranとの相違について述べる。そして、実際のプログラムをするために知っておくべき事項（JDKのインストール、テキストエディタの使用法、ソースコードの入力方法、HTMLファイルについて、 Javaのコンパイルと実行方法など）を解説する。

第2章では、Java の文法について、数値計算に必要な機能に重点をおいて解説する。単なる文法解説ではなく、実際のプログラム例を使用して理解しやすい説明をおこなう。まず、コンソールへの結果の出力命令を説明し、以降はそれを用いて命令の実行結果を表示する。また、アプレットおよびスタンドアローンアプリケーションについても、数値計算の結果表示に有用なことがらに絞って解説する。次に数値と演算子を使用して計算結果を出力し、種々の演算子やその優先順位を説明する。次に変数の定義と使用法、変数の型について述べる。また、オブジェクト指向やクラスなどは初学者が混乱しないように必要に応じて徐々に説明するスタイルとした。その後に、if、for、などの命令をひとつずつ説明し、予約語などについても解説していく。また、数値計算には欠かせない数学関数などを

含む Math クラスを説明し、加えてユーザーによる関数の定義と使用法について述べる。さらに、配列変数の宣言方法と使用例も示す。

第3章では、やや進んだ入門としてプログラムの開始時や実行中の数値の入力と、計算結果の表示などを説明する。また、Javaの特徴の一つであるオブジェクト思考についても簡単に説明する。

第4章～第6章では、初歩的な数値計算の問題をいくつか取り上げ、そのプログラムを示すと共に内容を解説する。予定する例題としては、数値微分、数値積分、方程式の解、ベクトル計算、行列計算、連立方程式の解などがある。

第7章、第8章では、さらに実用的な例題と、より高度な使用法について述べる。また、関数の再帰的呼出などについても解説する。例題としては、多桁計算、フーリエ級数、高速フーリエ変換、などを予定している。さらに、Excel など他のアプリケーションとの連携法について解説し、結果のファイルへの出力法も示す。

また、付録として、Java の簡易なリファレンス、主なエラーメッセージとその対策、バージョンやブラウザによる互換性、WEBでの情報源などについて述べている。

用例も示す。

第3章では、やや進んだ入門としてプログラムの開始時や実行中の数値の入力と、計算結果の表示などを説明する。また、Javaの特徴の一つであるオブジェクト思考についても簡単に説明する。

第4章～第6章では、初歩的な数値計算の問題をいくつか取り上げ、そのプログラムを示すと共に内容を解説する。予定する例題としては、数値微分、数値積分、方程式の解、ベクトル計算、行列計算、連立方程式の解などがある。

第7章、第8章では、さらに実用的な例題と、より高度な使用法について述べる。また、関数の再帰的呼出などについても解説する。例題としては、多桁計算、フーリエ級数、高速フーリエ変換、などを予定している。さらに、JavaScript の制限をこえた利用法として、Java との連携によるグラフの作成、Excel など他のアプリケーションとの連携法について解説する。

また、付録として、JavaScript の簡易なリファレンス、主なエラーメッセージとその対策、C/C++ や Java への移行について、バージョンやブラウザによる互換性についてなどについて述べる。